O'REILLY®



原书第2版

利用Python进行数据分析

Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython



powered by



Wes McKinney 著

徐敬一 译



机械工业出版社
China Machine Press

VISIT...

LANZAROTE Caliente.COM

O’Reilly精品图书系列

利用Python进行数据分析（原书第2版）

Python for Data Analysis：Data Wrangling with Pandas，NumPy，and IPython，2nd Edition

（美）韦斯·麦金尼（Wes McKinney） 著

徐敬一 译

ISBN：978-7-111-60370-2

本书纸版由机械工业出版社于2018年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）在中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾地区）制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

译者序
前言
第1章　准备工作
  1.1　本书内容
    1.1.1　什么类型的数据
  1.2　为何利用Python进行数据分析
    1.2.1　Python作为胶水
    1.2.2　解决“双语言”难题
    1.2.3　为何不使用Python
  1.3　重要的Python库
    1.3.1　NumPy
    1.3.2　pandas
    1.3.3　matplotlib
    1.3.4　IPython与Jupyter
    1.3.5　SciPy
    1.3.6　scikit-learn
    1.3.7　statsmodels
  1.4　安装与设置
    1.4.1　Windows
    1.4.2　Apple（OS　X和macOS）
    1.4.3　GNU/Linux
    1.4.4　安装及更新Python包
    1.4.5　Python　2和Python　3
    1.4.6　集成开发环境和文本编辑器
  1.5　社区和会议
  1.6　快速浏览本书
    1.6.1　代码示例
    1.6.2　示例数据
    1.6.3　导入约定
    1.6.4　术语
第2章　Python语言基础、IPython及Jupyter　notebook
  2.1　Python解释器
  2.2　IPython基础
    2.2.1　运行IPython命令行
    2.2.2　运行Jupyter　notebook
    2.2.3　Tab补全
    2.2.4　内省
    2.2.5　%run命令

2.2.6　执行剪贴板中的程序
2.2.7　终端快捷键
2.2.8　关于魔术命令
2.2.9　matplotlib集成
2.3　Python语言基础
2.3.1　语言语义
2.3.2　标量类型
2.3.3　控制流
第3章　内建数据结构、函数及文件
3.1　数据结构和序列
3.1.1　元组
3.1.2　列表
3.1.3　内建序列函数
3.1.4　字典
3.1.5　集合
3.1.6　列表、集合和字典的推导式
3.2　函数
3.2.1　命名空间、作用域和本地函数
3.2.2　返回多个值
3.2.3　函数是对象
3.2.4　匿名（Lambda）函数
3.2.5　柯里化：部分参数应用
3.2.6　生成器
3.2.7　错误和异常处理
3.3　文件与操作系统
3.3.1　字节与Unicode文件
3.4　本章小结
第4章　NumPy基础：数组与向量化计算
4.1　NumPy　ndarray：多维数组对象
4.1.1　生成ndarray
4.1.2　ndarray的数据类型
4.1.3　NumPy数组算术
4.1.4　基础索引与切片
4.1.5　布尔索引
4.1.6　神奇索引
4.1.7　数组转置和换轴
4.2　通用函数：快速的逐元素数组函数
4.3　使用数组进行面向数组编程
4.3.1　将条件逻辑作为数组操作
4.3.2　数学和统计方法

4.3.3　布尔值数组的方法
4.3.4　排序
4.3.5　唯一值与其他集合逻辑
4.4　使用数组进行文件输入和输出
4.5　线性代数
4.6　伪随机数生成
4.7　示例：随机漫步
4.7.1　一次性模拟多次随机漫步
4.8　本章小结
第5章　pandas入门
5.1　pandas数据结构介绍
5.1.1　Series
5.1.2　DataFrame
5.1.3　索引对象
5.2　基本功能
5.2.1　重建索引
5.2.2　轴向上删除条目
5.2.3　索引、选择与过滤
5.2.4　整数索引
5.2.5　算术和数据对齐
5.2.6　函数应用和映射
5.2.7　排序和排名
5.2.8　含有重复标签的轴索引
5.3　描述性统计的概述与计算
5.3.1　相关性和协方差
5.3.2　唯一值、计数和成员属性
5.4　本章小结
第6章　数据载入、存储及文件格式
6.1　文本格式数据的读写
6.1.1　分块读入文本文件
6.1.2　将数据写入文本格式
6.1.3　使用分隔格式
6.1.4　JSON数据
6.1.5　XML和HTML：网络抓取
6.2　二进制格式
6.2.1　使用HDF5格式
6.2.2　读取Microsoft　Excel文件
6.3　与Web　API交互
6.4　与数据库交互
6.5　本章小结

第7章　数据清洗与准备
7.1　处理缺失值
7.1.1　过滤缺失值
7.1.2　补全缺失值
7.2　数据转换
7.2.1　删除重复值
7.2.2　使用函数或映射进行数据转换
7.2.3　替代值
7.2.4　重命名轴索引
7.2.5　离散化和分箱
7.2.6　检测和过滤异常值
7.2.7　置换和随机抽样
7.2.8　计算指标/虚拟变量
7.3　字符串操作
7.3.1　字符串对象方法
7.3.2　正则表达式
7.3.3　pandas中的向量化字符串函数
7.4　本章小结
第8章　数据规整：连接、联合与重塑
8.1　分层索引
8.1.1　重排序和层级排序
8.1.2　按层级进行汇总统计
8.1.3　使用DataFrame的列进行索引
8.2　联合与合并数据集
8.2.1　数据库风格的DataFrame连接
8.2.2　根据索引合并
8.2.3　沿轴向连接
8.2.4　联合重叠数据
8.3　重塑和透视
8.3.1　使用多层索引进行重塑
8.3.2　将“长”透视为“宽”
8.3.3　将“宽”透视为“长”
8.4　本章小结
第9章　绘图与可视化
9.1　简明matplotlib　API入门
9.1.1　图片与子图
9.1.2　颜色、标记和线类型
9.1.3　刻度、标签和图例
9.1.4　注释与子图加工
9.1.5　将图片保存到文件

9.1.6　matplotlib设置
9.2　使用pandas和seaborn绘图
9.2.1　折线图
9.2.2　柱状图
9.2.3　直方图和密度图
9.2.4　散点图或点图
9.2.5　分面网格和分类数据
9.3　其他Python可视化工具
9.4　本章小结
第10章　数据聚合与分组操作
10.1　GroupBy机制
10.1.1　遍历各分组
10.1.2　选择一列或所有列的子集
10.1.3　使用字典和Series分组
10.1.4　使用函数分组
10.1.5　根据索引层级分组
10.2　数据聚合
10.2.1　逐列及多函数应用
10.2.2　返回不含行索引的聚合数据
10.3　应用：通用拆分-应用-联合
10.3.1　压缩分组键
10.3.2　分位数与桶分析
10.3.3　示例：使用指定分组值填充缺失值
10.3.4　示例：随机采样与排列
10.3.5　示例：分组加权平均和相关性
10.3.6　示例：逐组线性回归
10.4　数据透视表与交叉表
10.4.1　交叉表：crosstab
10.5　本章小结
第11章　时间序列
11.1　日期和时间数据的类型及工具
11.1.1　字符串与datetime互相转换
11.2　时间序列基础
11.2.1　索引、选择、子集
11.2.2　含有重复索引的时间序列
11.3　日期范围、频率和移位
11.3.1　生成日期范围
11.3.2　频率和日期偏置
11.3.3　移位（前向和后向）日期
11.4　时区处理

11.4.1　时区的本地化和转换
11.4.2　时区感知时间戳对象的操作
11.4.3　不同时区间的操作
11.5　时间区间和区间算术
11.5.1　区间频率转换
11.5.2　季度区间频率
11.5.3　将时间戳转换为区间（以及逆转换）
11.5.4　从数组生成PeriodIndex
11.6　重新采样与频率转换
11.6.1　向下采样
11.6.2　向上采样与插值
11.6.3　使用区间进行重新采样
11.7　移动窗口函数
11.7.1　指数加权函数
11.7.2　二元移动窗口函数
11.7.3　用户自定义的移动窗口函数
11.8　本章小结
第12章　高阶pandas
12.1　分类数据
12.1.1　背景和目标
12.1.2　pandas中的Categorical类型
12.1.3　使用Categorical对象进行计算
12.1.4　分类方法
12.2　高阶GroupBy应用
12.2.1　分组转换和“展开”GroupBy
12.2.2　分组的时间重新采样
12.3　方法链技术
12.3.1　pipe方法
12.4　本章小结
第13章　Python建模库介绍
13.1　pandas与建模代码的结合
13.2　使用Patsy创建模型描述
13.2.1　Patsy公式中的数据转换
13.2.2　分类数据与Patsy
13.3　statsmodels介绍
13.3.1　评估线性模型
13.3.2　评估时间序列处理
13.4　scikit-learn介绍
13.5　继续你的教育
第14章　数据分析示例

14.1　从Bitly获取1.USA.gov数据
14.1.1　纯Python时区计数
14.1.2　使用pandas进行时区计数
14.2　MovieLens　1M数据集
14.2.1　测量评价分歧
14.3　美国1880～2010年的婴儿名字
14.3.1　分析名字趋势
14.4　美国农业部食品数据库
14.5　2012年联邦选举委员会数据库
14.5.1　按职业和雇主的捐献统计
14.5.2　捐赠金额分桶
14.5.3　按州进行捐赠统计
14.6　本章小结
附录A　高阶NumPy
附录B　更多IPython系统相关内容

译者序

2014年我刚走上工作岗位时，Python数据分析技术开始在中国流行，越来越多的人开始学习和使用这门技术。当时，Python的主流版本是2.7，3.4版本刚刚发布，但认可度和流行度并不高。很幸运，我阅读了本书的第1版。第1版基于Python 2.7，介绍了大量的实用技术，包括pandas、NumPy的使用等。在实践中，书中的技术被我大量应用，帮助我更快更好地处理了各类数据。

时过境迁，从本书英文版第1版2012年出版至今，已经过去了6年。这6年中，Python的主流版本从2.7升级到了3.6。无论是否情愿，大部分Pythoner都不得不学会适应新版本；而pandas则从0.1.0版本迭代到如今的0.22.0版本。版本号的持续增加意味着新技术、新特性的不断丰富。举例来说，将带有多层索引的数据透视表写入Excel在2015年之前是无法使用pandas完成的，在0.17版本后该功能被加入。因此，本书的第一版内容已经略显落后。

2017年10月下旬，本书作者Wes McKinney先生更新了本书的第2版。在第2版中，他将Python版本更新到3.6，介绍了pandas的一些新接口和功能，并新增了大量现实世界的数据分析实例，以确保本书的可实践性。我在2017年11月18日接到出版社的翻译邀请后便开始了翻译工作。McKinney先生的写作风格朴实、形象，因而我的翻译过程较为顺畅，但书中部分口语化的叙述也因为中英文表达方式的差异而增加了意译的难度。

在本书翻译过程中，我得到了很多帮助。首先要感谢华章公司的王春华编辑和冯秀泳编辑，他们在翻译过程中给了我耐心指导。在审稿时，来自国内Python圈的朋友们对本书进行了仔细而全面的审核，他们是网易数据工程师马喆诚、早稻田大学研究生梁塬、大疆工程师王波。感谢他们对本书的付梓而做出的贡献。此外，我还要感谢我的女朋友易慧娟，感谢你在生活中对我的各种好，我爱你！

为了让国内读者在第一时间读到这本畅销国外技术著作，出版社和我都加快了工作进度，但时间紧、任务重，再加之本人水平有限，翻译工作中难免会出现一些失误。欢迎读者将阅读过程中发现的问题发送至我的邮箱（xujingyi46@163.com）。

本书英文版的副书名是“Data Wrangling with Pandas，NumPy，and IPython”，其中Wrangling是一个很难直译的词汇，它的原意是争执、争论，但在书中它描述的是将数据进行规整、处理的意思。希望读者读完本

书后，可以使用好pandas、NumPy和IPython这些工具，更好地完成数据处理、分析的学习和工作。Enjoy it！

徐敬一

2018年5月于中国工商银行合肥后台中心

前言

第2版新内容

本书第1版出版于2012年，彼时基于Python的开源数据分析库（例如pandas）仍然是一个发展迅速的新事物。在本次更新、拓展的第2版中，我在一些章节内进行了修改，以解释过去5年中发生的不兼容的变更、弃用和一些新特性。此外，我还添加了新内容，用以介绍在2012年还不存在或者不成熟的工具。最后，我会避免把一些新兴的或者不太可能走向成熟的开源项目写入本书。我希望本版的读者能够发现本书内容在2020年或者2021年仍然几乎像在2017年一样适用。

第2版中的主要更新包括：

·所有的代码，包括把Python的教程更新到了Python 3.6版本（第1版中使用的是Python 2.7）

·更新了Python第三方发布版Anaconda和其他所需Python包的安装指引

·更新pandas库到2017年的最新版

·新增一章，关于更多高级pandas工具和一些使用提示

·新增statsmodels和scikit-learn的简明使用介绍

除了以上更新内容，我还重新组织了第1版的部分重要内容，使本书对新手来说更易于理解。

本书约定

以下印刷约定将在本书中使用：

斜体（Italic）

表示新的术语、URL、email地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序清单以及段落中的程序元素，例如变量名、函数名、数据库、数据类型、环境变量、表达式和关键字等。

等宽粗体（Constant width bold）

表示命令或其他应当由用户键入的文本。

等宽斜体（Constant width italic）

表示应当由用户提供的值来替代的文本，或者其他由上下文决定的值。

 本符号表示提示或建议。

 本符号表示一般性说明。

 本符号表示警告。

使用代码示例

可以通过本书的GitHub仓库获取本书每一章中的数据文件和相关材料。GitHub仓库地址：http://github.com/wesm/pydata-book 。

本书的目的在于帮助你完成工作。一般来说，本书提供的示例代码，你可以在你的程序或文档中使用而无须联系我们获取许可，除非你需要重造大量代码。举例来说，使用本书中的代码段编写程序无须授权许可，但销售或发行O'Reilly图书的CD-ROM代码示例则需要许可。引用本书代码回答问题不需要许可，但在你的产品文档中大量使用本书示例代码则需要许可。

我们鼓励注明资料来源的行为，但这并不是必需的。来源注明通常包括书名、作者、出版社及ISBN，例如：“Python for Data Analysis by Wes McKinney（O'Reilly）.Copyright 2017Wes McKinney，978-1-491-95766-0”。

如果你认为你对本书示例代码的使用超过了正常使用范围或者需要以上介绍的授权许可，请联系permissions@oreilly.com。

O'Reilly Safari

Safari（前身为Safari Books Online）是一个会员制的培训、参考网站，服务于企业、政府、教育者和个人。

会员可以访问数千书籍、培训视频、学习路径、交互教程和超过250家出版商的企划列表，包括O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley&Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones&Bartlett、Course Technology等。

更多信息，请访问http://oreilly.com/safari 。

如何联系我们

对于本书如果有任何意见或疑问，请按照以下地址联系本书出版商。

美国：

O'Reilly Media，Inc.

1005Gravenstein Highway North

Sebastopol，CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书准备了一个网页，用于陈列勘误、示例和其他附加信息。访问地址是：http://bit.ly/python_data_analysis_2e 。

针对本书评论或提出技术问题，请发送邮件至：bookquestions@oreilly.com 。

关于本书的更多信息、课程、会议及新闻，请访问我们的网站：http://www.oreilly.com 。

Facebook联系我们：http://facebook.com/oreilly

Twitter联系我们：http://twitter.com/oreillymedia

YouTube观看我们的视频：http://www.youtube.com/oreillymedia

致谢

本书是全世界很多人多年来富有成效的讨论、协作和支持的成果。我想对他们中的一些代表致以谢意。

怀念：John D.Hunter（1968——2012）

我们亲爱的朋友和同行John D.Hunter在经历了一场与结肠癌的战斗后，于2012年8月28日离开了世界。那时正是我完成本书第1版最终手稿后不久。

John对Python科学计算和数据社区的影响之大难以估量，他给我们留下的遗产价值非凡。除了在2000年初期开发matplotlib之外（那时Python还没有当下如此流行），他还帮助塑造了一代核心开源开发者的文化，如今这些开发者已经成为Python生态系统的顶梁柱，而Python生态系统对于现如今的我们来说似乎是理所当然的。

在2010年1月，我开源生涯的早期，那时候pandas刚刚发布了0.1版本，我便有幸结识了John。即便在最黑暗的时期，他的才华和指导仍在帮助我推动pandas前进，实现Python成为数据分析第一语言的愿景。

John与IPython、Jupyter项目的先锋Fernando Pérez、Brian Granger及其他很多Python社区的倡议人联系紧密。我们四人曾经希望共同写作一本书，但只有我个人时间最为自由，所以这个想法被搁置了。我非常确信他会为过去5年中我们个人及我们社区所取得的成就感到骄傲。

第2版致谢（2017）

距离我在2012年7月完成第1版手稿已经5年了。很多事情都发生了变化。Python社区获得了极大的成长，围绕Python的开源软件生态系统也十分繁荣。pandas核心开发者孜孜不倦的付出，使得pandas项目高速成长，也使得pandas的用户群体遍布Python数据科学生态系统的各个角落，没有他们本书将不会存在。pandas的核心开发者包括但不限于：Tom Augspurger、Joris van den Bossche、Chris Bartak、Phillip Cloud、gfyoung、Andy Hayden、Masaaki Horikoshi、Stephan Hoyer、Adam Klein、Wouter、Overmeire、Jeff Reback、Chang She、Skipper

Seabold、Jeff Tratner和y-p。

在第2版的实际写作过程中，非常感谢O'Reilly的工作人员在写作进程中给予的耐心帮助。他们是Marie Beaugureau、Ben Lorica和Colleen Toporek。我再次得到了优秀技术审阅人的支持，他们是Tom Augpurger、Paul Barry、Hugh Brown、Jonathan Coe和Andreas Müller。感谢你们。

本书的第1版已经被翻译成多种语言，包括汉语、法语、德语、日语、韩语和俄语。将本书翻译给外国读者，是一份工作量大且缺少关注的付出。感谢你们帮助全世界更多人士学会如何编程及使用数据分析工具。

在过去几年中，Cloudera和Two Sigma投资公司对我的持续开源开发工作的支持使我感到十分幸运。由于开源项目相对于用户基数的比例越来越小，向重要开源项目提供开发支持变得越来越重要。这是一件值得去做的正确工作。

第1版致谢（2012）

如果没有众多相关人士的支持，写作本书对我来说将会十分困难。

对于O'Reilly的工作人员，我非常感谢我的编辑Meghan Blanchette和Julie Steele，他们在整个写作过程对我给予指导。Mike Loukides还在建议阶段与我一起工作，帮助本书付梓。

我收到了大量相关人士丰富的技术审阅。尤其是Martin Blais和Hugh Brown，他们自始至终在提高本书示例的清晰度、组织度上提供了令人难以置信的帮助。James Long、Drew Conway、Fernando Pérez、Brian Granger、Thomas Kluyver、Adam Klein、Josh Klein、Chang She和Stéfan van der Walt每个人都审阅了本书的一章或多章，从很多角度提供了有效反馈。

我从数据社区的朋友和同行那里获得了很多关于示例和数据集的优秀想法，他们是：Mike Dewar、Jeff Hammerbacher、James Johndrow、Kristian Lum、Adam Klein、Hilary Mason、Chang She和Ashley Williams。

当然，我也非常感激开源科学Python社区的众多领头人，他们为我的开发工作打下了基础，并在本书写作过程中给予我鼓励：IPython核心团队（Fernando Pérez、Brian Granger、Min Ragan-Kelly、Thomas Kluyver和其他相关人士）、John Hunter、Skipper Seabold、Travis

Oliphant、Peter Wang、Eric Jones、Robert Kern、Josef Perktold、Francesc Alted、Chris Fonnesbeck以及其他由于人数太多而无法提及的人们。还有很多人士一直以来提供了支持、想法和鼓励：Drew Conway、Sean Taylor、Giuseppe Paleologo、Jared Lander、David Epstein、John Krowas、Joshua Bloom、Den Pilsworth、John Myles-White以及很多已经忘了姓名的人士。

我还要感谢前些年生活中的一些人。首先是在AQR的前同事，他们都曾为我在pandas方面的工作喝彩，他们是：Alex Reyfman、Michael Wong、Tim Sargen、Oktay Kurbanov、Matthew Tschantz、Roni Israelov、Michael Katz、Chris Uga、Prasad Ramanan、Ted Square和Hoon Kim，以及我的指导教授Haynes Miller（麻省理工学院）和Mike West（杜克大学）。

2014年，我在更新本书代码示例、修正一些由于pandas变更产生的错误时，从Phillip Cloud和Joris Van den Bossche处获得了重要帮助。

个人方面，感谢Casey在写作过程中为我提供了无价的日常支持，忍受我的情绪起伏直到我按计划表写出了最终手稿。最后，感谢我的父母Bill和Kim，他们教会我如何去追寻梦想、永不止步。

第1章　准备工作

1.1　本书内容

本书关注的是利用Python操作、处理、清洗和操作数据时的基本要点。我的目标是提供一份Python编程语言以及Python面向数据的类库生态系统和工具的指南，该指南将帮助你成为一个高效的数据分析师。尽管“数据分析”出现在书名里，但本书将明确专注于Python语言的编程、类库、工具而不是数据分析方法论。这就是你需要的Python数据分析编程。

1.1.1　什么类型的数据

当我说“数据”时，我想表达的准确含义是什么？主要的关注点是结构化数据，这个有意义的术语包含了众多常见的数据形式，例如：

·表格型的数据，每一列可能会包含不同的类型（字符串、数值、日期或其他）。这类数据包含了大部分类型的数据，它们通常存储在关系型数据库或者由制表符、逗号分隔的文本文件中。

·多维数组（矩阵）。

·由键位列关联的多张表数据（对于SQL用户来说就是主键或外键）。

·均匀或非均匀的时间序列。

以上是一份大致完整的清单。但该清单有时并不完全准确，很多数据集可以转换为一种更适合分析、建模的结构形式。如果不进行转换，从数据集中提取特征形成一种结构形式也是可行的。例如，一个新闻文章的数据集可以被处理为一个词频表，然后再用于情感分析。

大部分表格程序（比如微软Excel，或许是全世界应用最广泛的数据分析工具）的用户对这些类型的数据并不陌生。

1.2　为何利用Python进行数据分析

对很多人来说，Python编程语言拥有强大的吸引力。从1991年面世以来，Python与Perl、Ruby等语言成为最流行的解释型语言。Python和Ruby从2005年之后格外流行，可以基于众多web框架，比如Rails（Ruby）和Django（Python）进行网站搭建。此类语言通常称为脚本语言，因为它们可以用于快速编写小型程序、脚本或对其他任务进行自动化。我并不喜欢“脚本语言”这个术语，因为它的言外之意似乎是“脚本语言”无法构建大型软件。在解释型语言中，由于历史和文化上的原因，Python发展出了一个大型、活跃的科学计算及数据分析社区。在过去十年里，Python已经从一个最前沿或者说“后果自负”的科学计算语言，成为数据科学、机器学习和学术/工业界通用软件开发等领域最为重要的语言之一。

在数据科学、交互式计算以及数据可视化等领域，Python经常被拿来和其他开源或商业编程语言、工具进行对比，比如R、MATLAB、SAS、Stata等。近些年，Python提高了对类库的支持（比如pandas和scikit-learn），使得它成为数据分析任务的一个流行选择。再综合考虑Python在通用软件工程上的总体实力，它便成为搭建数据应用的首选语言。

1.2.1 Python作为胶水

Python在科学计算方面的成功部分是因为它很容易整合C、C++和FORTRAN等语言的代码。大部分现代计算环境都拥有相似的存量程序集，这些程序集使用FORTRAN和C的库进行线性代数、调优、积分、快速傅里叶变换等算法运算。很多公司和国家实验室都使用Python将过去数十年产生的存量软件黏合在一起。

很多程序中包含了一小部分运行起来要花费大量时间的代码，而大量“胶水代码”却很少运行。很多情况下，胶水代码的执行时间可以忽略不计；精力应当更多地投入优化计算瓶颈的方面，有些时候需要将代码迁移到底层语言，比如C。

1.2.2 解决“双语言”难题

在许多组织中，通常会使用一种更特殊的计算语言，比如用SAS或R针对想法进行研究、原型实现和测试，之后再将这些想法迁移到用Java、C#或C＋＋编写的大型生产环境上。逐渐地，人们发现Python不但更适用于研究和原型实现，也适合搭建生产系统。当一种语言可以满足需求时，为什么要维持两个开发环境呢？当使用相同程序工具集来兼顾研究人员和软件工程师的好处越发明显时，我认为会有越来越多的公司选择走使用一种语言的路线。

1.2.3 为何不使用Python

虽然Python可以为很多分析应用和通用系统提供一个优秀环境，但仍然有一些场景用Python并不合适。

由于Python是一个解释型语言，大多数情况下Python代码的运行效率会低于Java或C＋＋等编译型语言。因为开发者时间通常比CPU时间更有价值，很多人就愉快地选择了使用Python。然而，当需要一款低延迟、高资源利用要求的应用时（例如高频交易系统），为了尽可能获得最高性能，在底层语言（也意味着更低的代码生产力）比如C＋＋上花费编程时间将会更加值得。

当搭建高并发、多线程应用，尤其是多CPU绑定线程时，使用Python则会成为一项挑战。原因在于Python拥有全局解释器锁（GIL），这是一种防止解释器同时执行多个Python指令的机制。GIL存在的技术原因超出了本书的讨论范围。然而在大数据处理应用中，计算机集群需要在合理时间内处理一个数据集，单进程多线程系统的场景仍然是有需求的。

这并不是说Python无法执行真正的多线程、并行代码。Python的C语言拓展使用本地多线程（在C或C＋＋中）以并行方式运行代码，而不受GIL的影响，因为这些拓展通常无须与Python对象交互。

1.3 重要的Python库

对于那些对Python数据生态系统以及本书所使用的类库不太熟悉的人士，我将简要地介绍一部分重要的库。

1.3.1 NumPy

NumPy（http://numpy.org ）是Numerical Python的简写，是Python数值计算的基石。它提供多种数据结构、算法以及大部分涉及Python数值计算所需的接口。NumPy还包括其他内容：

·快速、高效的多维数组对象ndarray

·基于元素的数组计算或数组间数学操作函数

·用于读写硬盘中基于数组的数据集的工具

·线性代数操作、傅里叶变换以及随机数生成

·成熟的C语言API，允许Python拓展和本地的C或C＋＋代码访问NumPy的数据结构和计算设施。

除了NumPy赋予Python的快速数组处理能力之外，NumPy的另一个主要用途是在算法和库之间作为数据传递的数据容器。对于数值数据，NumPy数组能够比Python内建数据结构更为高效地存储和操作数据。此外，用底层语言编写的库，例如用C或Fortran编写的库，可以在NumPy数组存储的数据上直接操作，而无须将数据复制到其他内存中后再操作。因此，许多Python的数值计算工具将NumPy数组作为基础数据结构，或与NumPy进行无缝互操作。

1.3.2 pandas

pandas（http://pandas.pyda ta.org ）提供了高级数据结构和函数，这些数据结构和函数的设计使得利用结构化、表格化数据的工作快速、简单、有表现力。它出现于2010年，帮助Python成为强大、高效的数据分析环境。在本书中主要使用的pandas对象是DataFrame，它是用于实现表格化、面向列、使用行列标签的数据结构；以及Series，一种一维标签数组对象。

pandas将表格和关系型数据库（例如SQL）的灵活数据操作能力与NumPy的高性能数组计算的理念相结合。它提供复杂的索引函数，使得数据的重组、切块、切片、聚合、子集选择更为简单。由于数据操作、预处理、清洗在数据分析中是重要的技能，pandas将是本书的重要主题。

介绍一点背景知识，早在2008年，我在一家量化投资企业——AQR资本管理公司供职时，便开始了pandas的开发。那时候，我有一些独特的需求是工具清单上任何单个工具无法满足的：

·带有标签轴，支持自动化或显式数据对齐功能的数据结构——这可以防止未对齐数据和不同数据源的不同索引数据所引起的常见错误

·集成时间序列函数功能

·能够同时处理时间序列数据和非时间序列数据的统一数据结构

·可以保存元数据的算术操作和简化

·灵活处理缺失数据

·流行数据库（例如基于SQL的数据库）中的合并等关系型操作

我想将以上的工作在同一个地方完成，最好还能在一个拥有通用软件开发能力的语言中实现。Python就是一个很好的备选项，但是那时候并没有这类数据结构的整合集，也没有能提供相关功能的工具。结果就是pandas最初被开发出来用于解决金融和商业分析问题，pandas尤其擅长深度时间序列和处理商业进程中产生的时间索引数据。

使用R语言进行统计计算的用户对DataFrame的名称会非常熟悉，因为这个对象是根据相似的R data.frame对象进行命名的。与Python不同的是，数据框在R语言中是标准库中的内容。因此，pandas中的很多特征通

常与R核心的实现或者R的附加库提供的功能一致。

pandas的名字的来源是panel data，这是计量经济学中针对多维结构化数据集的术语。pandas也是Python data analysis（Python数据分析）自身的简写短语。

1.3.3 matplotlib

matplotlib（http://matplotlib.org ）是最流行的用于制图及其他二维数据可视化的Python库。它由John D.Hunter创建，目前由一个大型开发者团队维护。matplotlib被设计为适合出版的制图工具。对于Python编程者来说也有其他可视化库，但matplotlib依然使用最为广泛，并且与生态系统的其他库良好整合。我认为将它作为默认可视化工具是一个安全的选择。

1.3.4 IPython与Jupyter

IPython项目（http://ipython.org ）开始于2001年，由Fernando Pérez发起，旨在开发一个更具交互性的Python解释器。在过去的16年中，它成为Python数据技术栈中最重要的工具之一。尽管它本身并不提供任何计算或数据分析工具，它的设计侧重于在交互计算和软件开发两方面将生产力最大化。它使用了一种执行-探索工作流来替代其他语言中典型的编辑-编译-运行工作流。它还提供针对操作系统命令行和文件系统的易用接口。由于数据分析编码工作包含大量的探索、试验、试错和遍历，IPython可以使你更快速地完成工作。

2014年，Fernando和IPython团队发布了Jupyter项目（http://jupyter.org ）。Jupyter项目旨在设计一个适用于更多语言的交互式计算工具。IPython web notebook则成为Jupyter notebook，可以支持超过40种编程语言。IPython系统目前可以作为一个内核（一种编程语言模式）用于在Jupyter中使用Python。

IPython自身已成为Jupyter开源项目中的一个组件，后者提供交互性、探索性的高效环境。IPtyhon最古老、最简单的“模式”就是一个加强版的Python命令行，用于提高编写、测试、调试Python代码的速度。你也可以通过基于Web、支持多语言的代码“笔记本”——Jupyter Notebook来使用IPython系统。IPython命令行和Jupyter notebook对于数据探索和可视化非常有用。

Jupyter notebook系统允许你使用Markdown和HTML创建包含代码和文本的富文档。其他编程语言也针对Jupyter实现了内核，允许你在Jupyter中使用多种语言而不仅仅是Python。

对我个人来说，IPython涉及我工作的大部分内容，包括运行、调试、测试代码。

在随书资料（http://github.com/wesm/pydata-book ）中，你可以找到包含本书所有章节示例代码的Jupyter notebook。

1.3.5 SciPy

SciPy（http://scipy.org ）是科学计算领域针对不同标准问题域的包集合。以下是SciPy中包含的一些包：

scipy.integrate

数值积分例程和微分方程求解器

scipy.linalg

线性代数例程和基于numpy.linalg的矩阵分解

scipy.optimize

函数优化器（最小化器）和求根算法

scipy.signal

信号处理工具

scipy.sparse

稀疏矩阵与稀疏线性系统求解器

scipy.special

SPECFUN的包装器。SPECFUN是Fortran语言下实现通用数据函数的包，例如gamma函数。

scipy.stats

标准的连续和离散概率分布（密度函数、采样器、连续分布函数）、各类统计测试、各类描述性统计。

SciPy与NumPy一起为很多传统科学计算应用提供了一个合理、完整、成熟的计算基础。

1.3.6 scikit-learn

scikit-learn项目（http://scikit-learn.org ）诞生于2010年，目前已成为Python编程者首选的机器学习工具包。仅仅七年，scikit-learn就拥有了全世界1500位代码贡献者。其中包含以下子模块。

·分类：SVM、最近邻、随机森林、逻辑回归等

·回归：Lasso、岭回归等

·聚类：k-means、谱聚类等

·降维：PCA、特征选择、矩阵分解等

·模型选择：网格搜索、交叉验证、指标矩阵

·预处理：特征提取、正态化

scikit-learn与pandas、statsmodels、IPython一起使Python成了高效的数据科学编程语言。本书中并不会提供详细的scikit-learn指引，但我会简要介绍scikit-learn中的一些模型以及如何与本书中出现的其他工具一起使用这些模型。

1.3.7 statsmodels

statsmodels（http://statsmodels.org ）是一个统计分析包。它源自斯坦福大学统计学教授Jonathan Taylor利用R语言实现的各类分析模型。Skipper Seabold和Josef Perktold早在2010年便创建了新的statsmodels项目。自那之后该项目迅速成长，拥有大量活跃用户和贡献者者。Nathaniel Smith开发了Patsy项目，为R语言公式系统所驱动的statsmodels包提供公式、模型规范框架。

与scikit-learn相比，statsmodels包含经典的（高频词汇）统计学、经济学算法。它所包含的模型如下。

·回归模型：线性回归、通用线性模型、鲁棒线性模型、线性混合效应模型等

·方差分析（ANOVA）

·时间序列分析：AR、ARMA、ARIMA、VAR等模型

·非参数方法：核密度估计、核回归

·统计模型结果可视化

statsmodels更专注于统计推理，提供不确定性评价和p值参数。相反，scikit-learn更专注于预测。

和scikit-learn一样，我将简要介绍如何与NumPy和pandas一起使用statsmodels。

1.4 安装与设置

由于每个人使用Python的应用场景不一样，设置Python、安装附加包并没有一个统一的解决方案。很多读者并没有一个适合本书后续内容的Python开发环境，因此我将给出一份各操作系统上的详细Python安装说明。我推荐使用免费的Anaconda发布版。在本书写作的时候，Anaconda提供Python 2.7和Python 3.6两个版本，当然未来某个时间版本会变更。本书使用Python 3.6版本，我推荐使用Python 3.6或更高版本。

1.4.1 Windows

在Windows上起步，需要先下载Anaconda安装器（http://anaconda.com/downloads ）。我推荐读者按照Anaconda官网下载页上的安装说明进行安装，当读者读至此时安装说明可能会和本书出版时的说明不太一样。

现在，让我们来确定所有设置是否都正确。鼠标右击开始菜单，选择命令行，打开命令行应用（也就是cmd.exe）。通过输入python来启动Python解释器。你应该可以看到符合你下载的Anaconda版本的信息：

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:1
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

要退出命令行，按下Ctrl-D（Linux或macOS），按下Ctrl-Z（Windows），或者输入命令exit（）后按下回车键。

1.4.2 Apple（OS X和macOS）

下载OS X版的Anaconda安装器，命名如Anaconda3-4.1.0-MacOSX-x86_64.pkg。双击.pkg文件运行安装器。安装器运行时，会自动将Anaconda执行路径添加到你的.bash_profile文件中，该文件位于/Users/$USER/.bash_profle。

确认安装正常，可以尝试在系统命令行（打开终端应用得到命令提示符）中运行IPython：

```
$ ipython
```

要退出命令行，按下Ctrl-D，或者输入命令exit（）后按下回车键。

1.4.3 GNU/Linux

Linux下的安装细节取决于你所用的Linux版本，但是我会给出部分Linux系统的Python安装细节，例如Debian、Ubuntu、CentOS和Fedora。设置方法与OS X大致相同，Anaconda的安装方法除外。安装器是一段shell脚本，必须通过终端执行。选择x86（32位）还是x86_64（64位）安装器取决于你的操作系统是32位还是64位。安装时，你需要使用一个文件名类似于Anaconda3-4.1.0-Linux-x86_64.sh的文件，并在bash命令行输入：

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

部分Linux发行版包含所有所需的Python包，可以通过像apt这样的工具进行安装。本书的安装是针对Anaconda的。这两种方式下都很容易进行跨版本重现和包版本更新。

在接受许可后，需要选择Anaconda的安装路径。推荐在主目录下的默认位置进行安装——例如/home/$USER/anaconda（$USER一般就是你的用户名）。

Anaconda可能会问你是否需要将它的bin/目录添加到$PATH变量中去。如果在安装后出现了问题，你可以自行修改.bashrc（或者.zshrc，如果你使用的是zsh命令行）：

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

完成修改后，你可以新开一个终端进程或通过source~/.bashrc再次执行你的.bashrc。

1.4.4 安装及更新Python包

在本书阅读中，你可能想要安装Anaconda并不包含的额外Python包。通常通过以下命令进行安装：

```
conda install package_name
```

如果不奏效你可以使用pip包管理工具进行安装：

```
pip install package_name
```

你还可以使用conda update命令来更新包：

```
conda update package_name
```

pip还支持通过--upgrade标识升级：

```
pip install --upgrade package_name
```

贯穿本书，你会有多次机会使用以上命令。



当你能够同时使用conda和pip进行包安装时，请不要尝试使用pip更新conda安装的包，否则可能会导致环境问题。当使用Anaconda或者Miniconda时，最好还是使用conda进行更新。

1.4.5 Python 2和Python 3

Python 3.x的第一个版本发布于2008年年底。它包含了大量与Python 2.x代码不兼容的变更。因为距离Python在1991年第一次发布已经过去了17年，鉴于长期以来的经验教训，创造一个“震撼”的Python 3版本是极好的。

2012年，大部分科学数据分析社区仍然在使用Python 2.x，因为当时很多包并没有完全兼容Python 3。因此，本书的第1版使用了Python 2.7。现如今，用户可以根据喜好在Python 2.x和Python 3.x进行选择，因为基本上两个版本都有全量的库支持。

然而，Python 2.x将在2020年结束开发周期（包括重要安全补丁），所以在Python 2.7下创建新项目并不是个好主意。因此，本书将使用Python 3.6，该稳定版部署更为广泛、支持更为友好。我们将Python 2.7称为“传统Python”，将Python 3.x简称为“Python”。我推荐你也这么做。

本书使用Python 3.6作为基础，你的Python版本可能比3.6更新，但本书的代码示例应该是向前兼容的。部分代码示例可能会有不同的运行结果，或者无法在Python 2.7中运行。

1.4.6 集成开发环境和文本编辑器

当被问及我的标准开发环境时，我会回答说“IPython加文本编辑器”。通常，我会在IPython或Jupyter notebook中写一段代码，然后迭代测试、调试。这种方式有助于在交互情况下操作数据，并可以通过肉眼确认特定数据集是否做了正确的事。像pandas和NumPy库都被设计为适合在命令行下使用。

然而，当开发软件时，一些用户可能倾向于使用功能更为丰富的集成开发环境（IDE），而不是功能相对简单的文本编辑器比如Emacs或Vim。下面介绍一些IDE：

·PyDev（免费），基于Eclipse平台的IDE

·PyCharm，Jetbrains公司开发（对商业用户收费，对开源用户免费）

·Python Tools for Visual Studio（适合Windows用户）

·Spyder（免费），Anaconda集成的IDE

·Komodo IDE（收费）

由于Python十分流行，大多数文本编辑器，比如Atom和Sublime Text 2都对Python有较好的支持。

1.5　社区和会议

除了网络搜索，科学、数据相关的Python邮件列表对于解决问题也非常有帮助。可以看看下列邮件列表：

·pydata：与数据分析和pandas相关的谷歌群组列表

·pystatsmodels：与statsmodels和pandas相关的问题

·scikit-learn邮件列表（scikit-learn@python.org）以及Python机器学习相关内容

·numpy-discussion：NumPy相关问题

·scipy-user：与SciPy或科学相关的Python问题

我有意不给出上述邮件列表的URL以免以后发生变更，这些URL很容通过互联网搜索找到。

每年全世界都会举办很多Python编程者会议。如果你想联系其他和你有共同爱好的Python编程人士，建议你在可能的情况下尝试参加一个会议。很多会议会为没有能力负担入场费或旅行费的人士提供经济支持。以下会议可供考虑：

·PyCon和EuroPython：北美和欧洲的两大主要Python会议

·SciPy和EuroSciPy：北美和欧洲面向科学计算的会议

·PyData：全世界范围内一系列区域性的会议，主题为数据科学和数据分析用例

·国际和地区性的PyCon会议（参见http://pycon.org 上的完整列表）

1.6 快速浏览本书

如果你从未使用Python编程，你需要在第2章和第3章上花些时间，这两章我提供了一份关于Python语言特征、IPython命令行和Jupyter notebook的教程。这些内容是本书后续章节所需的预备知识。如果已经有Python经验，你可以跳过这两章。

下一章，我会简单介绍NumPy的关键特性，并在附录A中提供高级的NumPy使用技术。之后，我会介绍pandas，并将本书后续内容集中于应用pandas、NumPy和matplotlib（可视化）进行数据分析的主题上。我已经尽可能地按照递增的方式来组织全书的内容，但在部分章节中偶尔还是会有些细微的交叉，比如一些还不需要引入概念的独立场景。

尽管读者们可能会有很多不同的工作目的，但工作任务大体上会分为以下几个部分。

与外部世界交互

读写各种格式的文件以及数据存储

准备

对分析数据进行清洗、处理、联合、正态化、重组、切片、切块和转换

转换

将数学或统计操作应用到数据集的分组上以产生新的数据集（例如通过分组参数对一张大表进行聚合）

建模和计算

将数据接入到统计模型、机器学习算法和其他计算工具上

演示

创建动态或静态的图形可视化或文字概述

1.6.1　代码示例

本书大部分代码示例会按照IPython或Jupyter notebook中的形式用输入In和输出Out展现：

```
In [5]: CODE EXAMPLE
Out[5]: OUTPUT
```

当你看到这样的代码示例时，其用意就是让你将示例代码输入到In区，然后按下回车键执行代码（Jupyter中按下shift-Enter），然后输出结果展现在Out区。

1.6.2　示例数据

每章的示例数据托管在GitHub仓库（http://github.com/wesm/pydata-book ）上。你可以在命令行中使用Git版本控制系统下载这些数据，也可以直接从GitHub网站上下载数据的zip打包文件。如果你遇到了问题，请在我个人网站（http://wesmckinney.com ）上获取关于如何获得本书资料的最新指引。

我已经尽了最大的努力保障这包含了复现示例所需的所有事项，但可能还有一些错误或遗漏。如果发现了错误或遗漏，请给我发邮件：book@wesmckinney.com 。报告本书错误的最佳途径位于O’Reilly网站上的勘误页（http://bit.ly/pyDataAnalysis_errata ）。

1.6.3 导入约定

Python社区对一些常用模块采用了命名约定：

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

这意味着，当你看到np.arrange时，它引用的是NumPy中的arrange函数。这是因为一次性从像NumPy这样的大包中引入所有内容（from numpy import*）在Python软件开发中被认为是拙劣实践。

1.6.4 术语

我会使用编程和数据科学中的一些术语，这些术语你可能不熟悉，以下是一些简单的定义。

处理/处置/规整（munge/munging/wrangling）

描述的是将非结构化或者同时又很凌乱的数据整理成结构化、清晰形式的整个过程。时下，这个词在很多数据骇客中流传。在英文中，“Munge”（处理）和“grunge”（垃圾）谐音。

伪代码

用一种类似代码的形式描述算法或者过程，而事实上又不是实际有效的源代码。

语法糖

并不增加新特性，但便利于代码编写的编程语法。

第2章　Python语言基础、IPython及Jupyter notebook

2011年到2012年，我写了本书的第1版，当时并没有多少Python数据分析资源。这是个蛋生鸡、鸡生蛋的问题：很多我们现在觉得理所当然的库，比如pandas、scikit-learn和statsmodels在当时并不成熟。2017年，出现了大量关于数据科学、数据分析以及机器学习的文献，补充了先前仅面向计算机科学家、物理学家和其他研究领域的专业人员的通用科学计算工作。此外，还出现了大量非常优秀的书籍，这些书主要是关于Python编程语言自身以及如何成为高效的Python软件工程师。

本书是介绍如何使用Python处理数据的，我认为独立地概述一下Python内建数据结构的特性以及数据操作方面的库是很有必要的。因此，本章及第3章将介绍一些基本信息和知识，这些信息足以确保你读懂本书的后续章节。

在我看来，在Python中高效地分析数据并不需要完全精通如何利用Python语言开发软件。推荐使用IPython命令行和Jupyter notebook来实验代码示例，以及探索各种类型、函数和方法的文档。尽管我已经尽量按照增量方式来展现书中的内容，但可能还会偶尔遇到一些没有完全介绍的内容。

本书的大部分内容是关于如何基于数据表进行分析以及用于大型数据集的数据准备工具。为了使用这些工具，通常必须先把凌乱数据整理为更好看的（或者说更结构化的）表格形式。幸运的是，Python就是一个将数据快速规整为合理形式的理想语言。使用Python语言的能力越强，准备待分析数据集的工作就越轻松。

本书的一些工具最好是通过IPython或者Jupyter会话来探索。一旦学会了如何使用IPython和Jupyter来探索数据，我推荐你实验本书的示例并且可以再实验尝试一些别的内容。和其他键盘控制的命令行环境一样，练就常用命令的肌肉记忆也是学习曲线的一部分。



有一些Python中的概念在本章并未提及，比如类和面向对象编程，你会发现这些概念其实在Python数据分析中也是有用的。

为了加深你的Python知识，建议通过Python官方教程或者一本优秀的通用Python编程书籍来补充本章没有介绍的内容。推荐的入门书籍包括：

·《Python Cookbook》（第3版），作者为David Beazley和Brian K.Jones（O’Reilly）

·《Fluent Python》，作者为Luciano Ramalho（O’Reilly）

·《Effective Python》[1]，作者为Brett Slatkin（Pearson）

[1] 本书中文版已由机械工业出版社引进出版，书号是：978-7-111-52355-0。—编辑注

2.1 Python解释器

Python是一种解释型语言。Python解释器通过一次执行一条语句来运行程序。标准的交互式Python解释器可以通过在命令行输入python命令来启动：

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:1
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more informatio
>>> a = 5
>>> print(a)
5
```

你在命令行中看到的＞＞＞提示符是你键入代码的地方。要退出Python解释器回到命令行提示符，可以输入exit（）或者按下Ctrl-D。

通过Python命令，再把.py文件作为第一个参数就可以非常方便地运行Python程序。假设我们已经写好了一个叫作hello_world.py的文件：

```
print('Hello world')
```

可以执行以下命令去运行程序（hello_world.py必须在命令行的当前路径下）：

```
$ python hello_world.py
Hello world
```

虽然一些Python编程者通过这种方式执行他们所有的代码，但是那些做数据分析或科学计算的人士则会使用IPython和Jupyter notebook。IPython是一个加强版的Python解释器，Juypyter notebook是一种基于Web的代码笔记本，最初也是源于IPython项目。我在本章会介绍如何使用IPython和Jupyter，并在附录A中深入介绍IPython的功能。当你使用%run命令时，IPython会在同一个进程内执行指定文件中的代码，确保你在执行完成时可以立即探索结果。

```
$ ipython
```

```
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:1
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra detail

In [1]: %run hello_world.py
Hello world

In [2]:
```

默认的IPython提示符采用In[2]：风格的显示，而不是标准的＞＞＞提示符。

2.2　IPython基础

本节，我们将带你运行IPython命令行和Jupyter notebook，并介绍一些核心概念。

2.2.1 运行IPython命令行

你可以像启动标准Python解释器那样，通过ipython命令启动IPython命令行：

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017, 23:1
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra detail

In [1]: a = 5

In [2]: a
Out[2]: 5
```

你可以将Python语句输入命令行，然后按下回车键运行。当你在IPython中仅输入一个变量名，它会返回一个表示该对象的字符串：

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(7)}

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

前两行是Python语句，其中第二行创建了一个名为data的变量，并引用了新建的Python字典。最后一行在控制台中打印了data的值。

与常见的print打印语句不同，IPython中大多数Python对象被格式化为更可读、更美观的形式。如果使用print方法在标准Python解释器中打印data变量，可读性会差一些：

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print(data)
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.29048970536512
6: 0.3308507317325902}
```

IPython还提供执行任意代码块（通过复制粘贴实现）和整个Python脚本的功能。你也可以使用Jupyter notebook来处理大段的代码块，后续内容会有介绍。

2.2.2　运行Jupyter notebook

Jupyter项目中的主要组件就是notebook，这是一种交互式的文档类型，可以用于编写代码、文本（可以带标记）、数据可视化以及其他输出。Jupyter notebook与内核交互，内核是编程语言的交互式计算协议的实现。Python的Jupyter内核使用IPython系统进行内部活动。

需要启动Jupyter时，可以在终端中运行jupyter notebook命令：

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and sh
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

在很多平台上，Jupyter会自动打开你的默认网络浏览器（除非你使用了--no-browser命令）。你可以通过http地址来浏览notebook，地址是http://localhost:8888/ 。图2-1是谷歌Chrome浏览器中的效果。

 很多人使用Jupyter作为本地计算环境，但它其实也可以部署在服务器端，远程访问。我不会介绍这些内容，但如果有需要的话，建议你在网上搜索相关的主题。

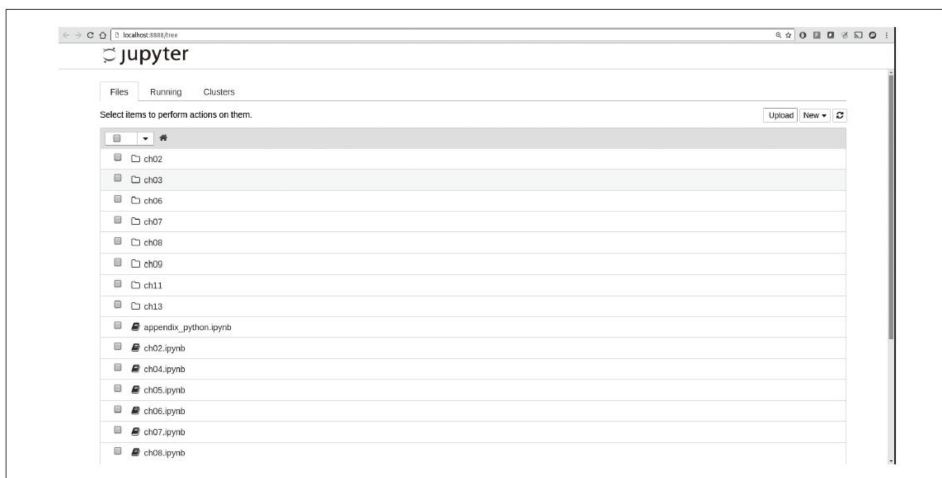


图2-1：Jupyter notebook登录页面

点击新建按钮选择“Python 3”或者“conda[default]”即可新建一个笔记本，然后你应该就可以看到如图2-2所示的内容。如果这是你第一次使用，请尝试点击空的代码“单元”，输入一行Python代码，然后按下Shift-Enter来执行。

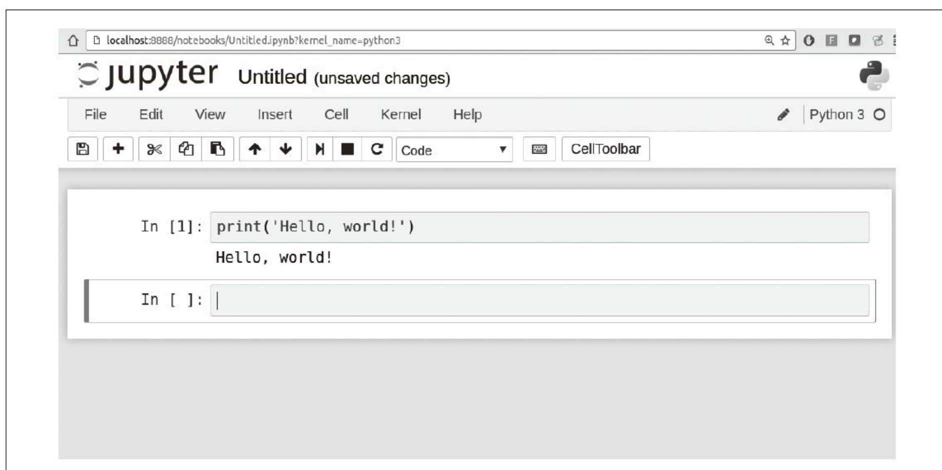


图2-2：Jupyter新建笔记本视图

当你保存笔记本的时候（在File菜单下有“Save and Checkpoint”选项），会自动生成一个后缀名为.ipynb的文件。这种文件格式会包含笔记本中当前的所有内容（包括已经产生的代码输出）。这些内容可以被其他

的Jupyter用户载入、编辑。要载入一个已经存在的笔记本，可以将文件放置在你启动命令行进程的路径下（或者是该路径下的子文件夹中），然后在登录页面双击文件名。你可以试试我在GitHub上的笔记本仓库wesm/pydata-book。见图2-3。

尽管Jupyter notebook提供了与IPython命令行不同的体验，但几乎所有的命令和工具都可以在两种环境下使用。

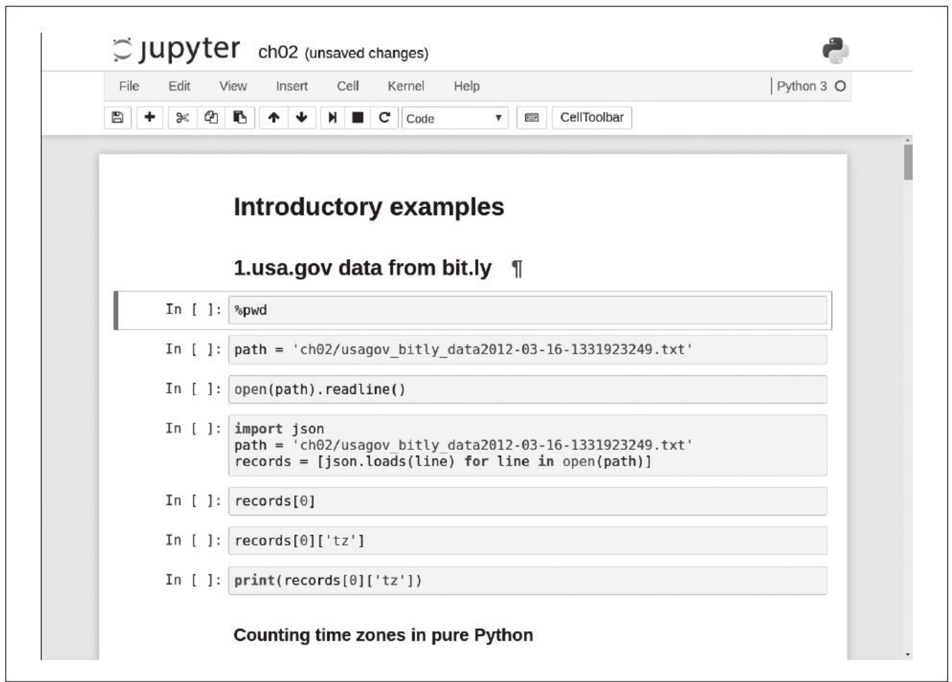


图2-3：通过Jupyter打开已经存在的笔记本

2.2.3 Tab补全

从表面上看，IPython只是看起来与标准Python解释器（通过python命令启动）有一些装饰性的区别。相较于标准Python命令行，IPython的提升之一就是tab补全功能，通常该功能在IDE或者其他交互式计算分析环境中才有。当在命令行输入表达式时，按下Tab键即可为任意变量（对象、函数等）搜索命名空间，与你目前已输入的字符进行匹配：

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and          an_example  any
```

在上述示例中，请注意IPython同时列出了我已定义的两个变量、关键字and和内建函数any。当然，你还可以在输入英文的句号之后，按下tab，对方法、属性的名称进行补全：

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append  b.count   b.insert  b.reverse
b.clear   b.extend  b.pop     b.sort
b.copy    b.index   b.remove
```

模块也可以通过相同的方式补全：

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date          datetime.MAXYEAR       datetime.timedelta
datetime.datetime      datetime.MINYEAR       datetime.timezone
datetime.datetime_CAPI datetime.time          datetime.tzinfo
```

在Jupyter notebook和新版的IPython（5.0及以上）中，自动补全是在下拉选项中展现，而不是文本输出。

 请注意IPython默认情况下隐藏了以下划线开始的方法和属性，诸

如魔术方法、内部“私有”方法和属性，以避免杂乱的显示（使新手混淆）。这些你当然也是可以使用tab补全的，但是必须先输入下划线才能看到它们。如果你总是想在tab补全时直接看到它们，则需要修改IPython配置。参见IPython官方文档可以找到相关内容。

tab补全除了在搜索交互命名空间和补全对象或模块属性时有用，在很多其他上下文场景中也有用。当输入任意路径（甚至是Python字符串）时，按下Tab键将补全你的计算机文件系统中匹配你输入内容的值：

```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat

In [7]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/README
datasets/movielens/ratings.dat   datasets/movielens/users.dat
```

与%run命令搭配使用（参见2.2.5节），该功能将为你节省大量键盘输入。

tab补全的另一个应用场景是在函数的关键字参数（包含=号）中节约时间，见图2-4。

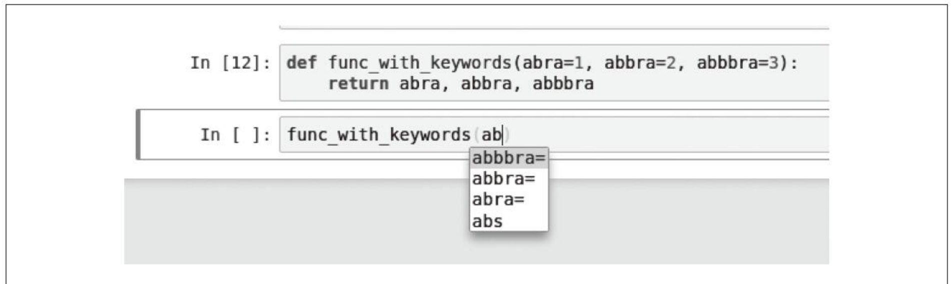


图2-4：在Jupyter notebook中自动补全函数关键字参数

本节之后将更进一步介绍函数。

2.2.4 内省

在一个变量名的前后使用问号（？）可以显示一些关于该对象的概要信息：

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:        list
String Form:[1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [10]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdou
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type:       builtin_function_or_method
```

这就是对象内省。如果对象是一个函数或实例方法且文档字符串已经写好，则文档字符串会显示出来。假设已经写好如下函数（你可以在IPython或Jupyter中复现）：

```
def add_numbers(a, b):
    """
    Add two numbers together
    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

然后使用？来显示文档字符串：

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
```

```
Docstring:
Add two numbers together

Returns
--------
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

使用双问号？？可以显示函数的源代码：

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    --------
    the_sum : type of arguments
    """
    return a + b
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

？有一个终极用途，可以像标准Unix或Windows命令行一样搜索IPython命名空间。把一些字符和通配符（星号*）结合在一起，会显示所有匹配通配符表达式的命名。例如，我们可以得到NumPy顶层函数中包含load的函数名列表：

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

2.2.5 %run命令

可以在IPython会话中使用%run命令运行任意的Python程序文件。假设你已经在ipython_script_test.py中写好了如下的简单脚本：

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

可以将文件名作为参数传给%run命令：

```
In [14]: %run ipython_script_test.py
```

脚本是在空白命名空间（没有导入模块或其他定义变量）中运行的，因此这个行为与在命令行中使用python script.py来运行程序是相同的。文件中定义的所有变量（导入的、函数中的、全局定义的）在运行后，可以在IPython命令行中使用（除非出现某种异常）：

```
In [15]: c
Out [15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

如果一个Python脚本需要命令行提供参数（通过sys.argv获得），那么则需要在命令行的文件路径后面加上参数进行传递：

如果你想让待运行的脚本使用交互式IPython命名空间中已有的变量，请使用%run-i替代普通的%run命令。

在Jupyter notebook中，如果你想将脚本导入一个代码单元，可以使用%load魔术函数：

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z

    a = 5
    b = 6
    c = 7.5
    result = f(a, b, c)
```

2.2.5.1　中断运行中的代码

在任意代码运行时按下Ctrl-C，无论脚本是通过%run或是其他长命令运行的，都将引起KeyboardInterrupt。除了某些特殊情况，这将导致所有的Python程序立即停止运行。


当一段Python代码被其他已经编译的扩展模块调用时，按下Ctrl-C并不会让程序立即停止运行。在这些情况下，你需要等到控制流重新返回Python解释器，在更糟糕的情况下可能要强制结束Python进程。

2.2.6 执行剪贴板中的程序

如果你正在使用Jupyter notebook，你可以将代码复制粘贴到代码单元，然后运行。在IPython中可以直接运行剪贴板中的程序。假设你在其他的应用中写了如下代码：

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

使用以上代码，最简单的方法就是%paste和%cpaste魔术函数。%paste会获得剪贴板中的所有文本，并在命令行中作为一个代码块去执行：

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- 传入文本的结尾 --
```

%cpaste与之类似，只不过它会给出一个特殊的提示符，让你粘贴代码：

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

使用%cpaste，你可以自由地在执行代码前尽可能多地粘贴代码。你也许会想着使用%cpaste在执行前看下粘贴的代码，如果你发现粘贴的代

码有误，可以按下Ctrl-C来中断%cpaste提示符。

2.2.7 终端快捷键

IPython有很多用于浏览提示、查看历史命令的快捷键（Emacs文本编辑器和Unix bash命令行的使用者会很熟悉）。表2-1总结了最常用的快捷键。图2-5展示了部分快捷功能，比如光标移动。

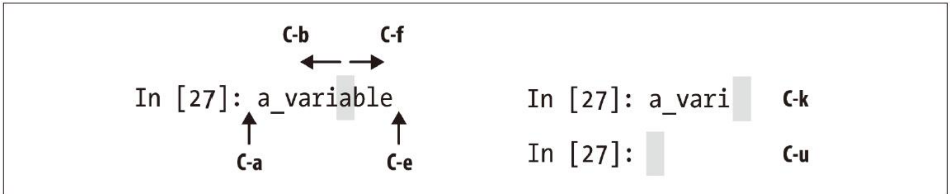


图2-5：IPython命令行的部分快捷键

表2-1：标准IPython快捷键

快捷键	描述
Ctrl-P 或 向上箭头	以当前输入内容开始，向后搜索历史命令
Ctrl-N 或 向下箭头	以当前输入内容开始，向前搜索历史命令
Ctrl-R	按行读取的反向历史搜索（部分匹配）
Ctrl-Shift-V	从剪贴板粘贴文本
Ctrl-C	中断当前正在执行的代码
Ctrl-A	将光标移动到本行起始位置
Ctrl-E	将光标移动到本行结束位置
Ctrl-K	删除光标后本行的所有内容
Ctrl-U	删除当前行
Ctrl-F	将光标向前移动一个字符
Ctrl-B	将光标向后移动一个字符
Ctrl-L	清除本屏内容

需要注意的是，Jupyter notebook有一个独立的快捷键集合用于导航和编辑。由于Jupyter notebook的快捷键更新比IPython更为频繁，建议你使用Jupyter notebook集成在菜单栏中的帮助系统。

2.2.8 关于魔术命令

IPython的特殊命令（没有内建到Python自身中去）被称为“魔术”命令。这些命令被设计用于简化常见任务，确保用户更容易控制IPython系统的行为。魔术命令的前缀符号是百分号%。例如，你可以使用%timeit来检查一段Python语句的执行时间，比如一个矩阵操作（后续将会详细讨论%timeit）：

```
In [20]: a = np.random.randn(100, 100)
In [20]: %timeit np.dot(a, a)
```

10000loops，best of 3：20.9魔术命令可以看作是IPython系统内部的命令行程序。大多数魔术命令都可以使用？查看额外的“命令行”选项：

```
In [21]: %debug?
Docstring:
::

  %debug [--breakpoint FILE:LINE] [statement [statement ...]]

Activate the interactive debugger.

This magic command support two ways of activating debugger.
One is to activate debugger before executing code.  This way, you
can set a break point, to step through the code from the point.
You can use this mode by giving statements to execute and optionally
a breakpoint.

The other one is to activate debugger in post-mortem mode.  You can
activate this mode simply running %debug without any argument.
If an exception has just occurred, this lets you inspect its stack
frames interactively.  Note that this will always work only on the la
traceback that occurred, so you must call this quickly after an
exception that you wish to inspect has fired, because if another one

occurs, it clobbers the previous one.
If you want IPython to automatically do this on every exception, see
the %pdb magic for more details.

positional arguments:
  statement        Code to run in debugger. You can omit this in cell
                      magic mode.

optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                      Set break point at LINE in FILE.
```

魔术函数也可以不加百分号%就使用，只要没有变量被定义为与魔术函数相同的名字即可。这种特性被称为自动魔术，通过%automagic进行启用/禁用关。

一些魔术函数也像Python函数一样，其输出可以赋给一个变量：

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

由于IPython的文档可以在系统内访问，建议使用%quickref或者%magic探索所有的特殊命令。表2-2是在IPython中高效进行交互式计算和Python开发所常用的重要命令。

表2-2：IPython常用魔术命令

命令	描述
%quickref	显示 IPython 快速参考卡
%magic	显示所有可用魔术命令的详细文档
%debug	从最后发生报错的底部进入交互式调试器
%hist	打印命令输入（也可以打印输出）历史
%pdb	出现任意报错后自动进入调试器
%paste	从剪贴板中执行已经预先格式化的 Python 代码
%cpaste	打开一个特殊提示符，手动粘贴待执行的 Python 代码
%reset	删除交互式命名空间中所有的变量 / 名称
%page *OBJECT*	通过分页器更美观地打印显示一个对象
%run *script.py*	在 IPython 中运行一个 Python 脚本
%prun *statement*	使用 CProfile 执行语句，并报告输出
%time *statement*	报告单个语句的执行时间
%timeit *statement*	多次运行单个语句计算平均执行时间；在估算代码最短执行时间时有用
%who, %who_ls, %whos	根据不同级别的信息 / 详细程度，展示交互命名空间中定义的变量
%xdel *variable*	在 IPython 内部删除一个变量，清除相关的引用

2.2.9 matplotlib集成

IPython能在分析计算领域流行的原因之一，就是它和数据可视化、用户界面库（如matplotlib）的良好集成。即使你从未使用过matplotlib也不必担心，本书的后续章节会详细讨论该部分内容。%matplotlib魔术函数可以设置matplotlib与IPython命令行或Jupyter notebook的集成。这个命令很重要，否则你创建的图可能不会出现（notebook），或者直到结束也无法控制会话（命令行）。

在IPython命令行中，运行%matplotlib命令可以生成多个绘图窗口，而无须干扰控制台的会话。

```
In [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

在Jupyter中，命令会有些许不同（见图2-6）：

```
In [26]: %matplotlib inline
```

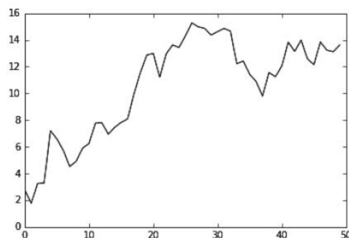


图2-6：Jupyter内联matplotlib绘图

2.3 Python语言基础

在本节，我会给出Python编程核心概念以及语言机制方面的概述。下一章，我会更为详细地介绍Python内建数据结构、函数以及其他内建工具。

2.3.1 语言语义

Python语言的设计非常独特，它侧重于可读性、易用性及清晰性。一部分人则认为它是“可执行的伪代码”。

2.3.1.1 缩进，而不是大括号

Python使用缩进（tab或者空格）来组织代码，而不是像其他语言比如R、C++、Java和Perl那样用大括号。考虑使用for循环来实现排序算法：

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

一个冒号代表一个缩进代码块的开始，单个代码块中所有的代码必须保持相同的缩进，直到代码块结束。

无论你喜欢或讨厌这种缩进，缩进就是Python编程者的真实生活。个人看法，缩进使Python的代码比我用过的其他语言更具可读性。刚开始看起来会不太适应，但很快你就会习惯。

我强烈推荐使用四个空格作为你的默认缩进，而不是用tab。很多文本编辑器可以设置用空格替代tab（请照做）。有些人使用tab或者其他数量的空格，但使用两个空格的并不常见。对大多数Python编程者来说，四个空格是标准形式，因此在非特殊原因下我推荐使用四个空格来缩进。

目前为止，你见到的Python语句都不是以分号结尾的。然而分号也是可以用于在一行内将多条语句之间进行分隔：

```
a = 5; b = 6; c = 7
```

我们不鼓励在Python中将多条语句写成一行，因为这样会使代码可读性下降。

2.3.1.2 一切皆为对象

Python语言的一个重要特征就是对象模型的一致性。每一个数值、字符串、数据结构、函数、类、模块以及所有存在于Python解释器中的事物，都是Python对象。每个对象都会关联到一种类型（例如字符串、函数）和内部数据。在实践中，一切皆为对象使得语言非常灵活，甚至函数也可以被当作对象来操作。

2.3.1.3 注释

所有写在#号之后的文本会自动被Python解释器忽略。因此通常使用#在代码中添加注释。有时候你会想排除部分代码但又不想删除，一个简单的解决办法就是把代码注释掉：

```
results = []
for line in file_handle:
    # 当前保持行为空
    # if len(line) == 0:
    #   则继续
    results.append(line.replace('foo', 'bar'))
```

注释也可以写在一行被执行代码的后面。部分编程者更习惯把注释写在特定的一行后面，这在很多时候有用：

```
print("Reached this line")  # 简单的状态报告
```

2.3.1.4 函数和对象方法的调用

调用函数时，向函数括号里传递0或多个参数，通常会把返回值赋值给一个变量：

```
result = f(x, y, z)
g()
```

几乎所有的Python对象都有内部函数，称为方法，可以访问到对象内部的内容。你可以通过以下语法调用它们：

```
obj.some_method(x, y, z)
```

函数传参既可以是位置参数也可以是关键字参数：

```
result = f(a, b, c, d=5, e='foo')
```

之后会详细介绍Python的参数机制。

2.3.1.5　变量和参数传递

在Python中对一个变量（或者变量名）赋值时，你就创建了一个指向等号右边对象的引用。实际上，当有一个整数列表时：

```
In [8]: a = [1, 2, 3]
```

假设我们将a赋值给一个新的变量b：

```
In [9]: b = a
```

在某些语言中，会是数据[1，2，3]被拷贝的过程。在Python中，a和b实际上是指向了相同的对象，即原来的[1，2，3]（在图2-7中示范）。你可以通过向a中添加一个元素，然后检查b来证明：

```
In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```

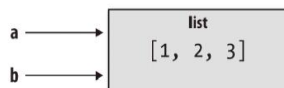


图2-7：两个引用指向同一个对象

理解Python引用语义中复制数据的时机、方法和原因的机制，对于利用Python处理大数据集尤其重要。

赋值也被称为绑定，这是因为我们将一个变量名绑定到了一个对象上。已被赋值的变量名有时也会被称为被绑定变量。

当你将对象作为参数传给一个函数时，指向原始对象的新的本地变量

就会被创建而无须复制。如果你将一个新的对象绑定到一个函数内部的变量上，这种变更不会在上级范围中产生影响。因此，更换可变参数的内部值是可以做到的。假设我们有如下函数：

```
def append_element(some_list, element):
    some_list.append(element)
```

之后我们可以得到以下结果：

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

2.3.1.6 动态引用、强类型

与Java、C++等大多数编译型语言不同，Python中的对象引用并不涉及类型。以下操作是没有问题的：

```
In [12]: a = 5

In [13]: type(a)
Out[13]: int

In [14]: a = 'foo'

In [15]: type(a)
Out[15]: str
```

变量对于对象来说只是特定命名空间中的名称；类型信息是存储在对象自身之中的。一些人可能会急于将Python总结为“非类型化语言”。这并不准确，考虑以下示例：

```
In [16]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent cal
<ipython-input-16-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```

在某些语言中，例如Visual Basic中，字符串'5'可能会隐式地转换为整数，然后得到10。而在另一些语言中，例如JavaScript，数字5可能会转换成字符串，生成一个结合字符串'55'。在这里，Python被认为是强类型语言，这意味着所有的对象都拥有一个指定的类型（或类），隐式的转换只在某些特定、明显的情况下发生：

```
In [17]: a = 4.5

In [18]: b = 2

# 字符串格式化，用于后面访问
In [19]: print('a is {0}, b is {1}'.format(type(a), type(b)))
a is <class 'float'>, b is <class 'int'>
In [20]: a / b
Out[20]: 2.25
```

了解对象的类型是非常重要的，写出可以处理多种不同输入的函数是非常有用的。你可以使用isinstance函数来检查一个对象是否是特定类型的实例。

```
In [21]: a = 5

In [22]: isinstance(a, int)
Out[22]: True
```

isinstance接受一个包含类型的元组，你可以检查对象的类型是否在元组中的类型中：

```
In [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: True
```

2.3.1.7　属性和方法

Python中的对象通常都会有属性（Python对象“内部”存储的其他对象）和方法（与对象内部对象有关的函数，相关的对象可以连接到对象内部数据）。属性和方法都可以通过形如obj.attribute_name的语法进行调用：

```
In [1]: a = 'foo'

In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

属性和方法也可以通过getattr函数获得：

```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

在其他的语言中，通过变量名访问对象通常被称为“反射”。虽然本书中我们不会广泛使用getattr以及相关的hasattr和setattr函数，但它们可以用来高效地编写通用、可复用的代码。

2.3.1.8 鸭子类型

通常情况下你并不关心某个对象的具体类型，而是关心它是否拥有某个特殊的方法或行为。“鸭子类型”的说法源于“一个东西走起来像鸭子叫起来像鸭子，那它就是鸭子”。例如，你可以验证一个对象如果实现了迭代器协议，那它一定是可以迭代的。对于很多对象来说，它包含了一个__iter__魔术方法，但使用iter函数是一个更好的、独立的方法：

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # 不可遍历
        return False
```

对于绝大部分Python容器类型的对象，iter函数都会返回True：

```
In [29]: isiterable('a string')
Out[29]: True

In [30]: isiterable([1, 2, 3])
```

```
Out[30]: True

In [31]: isiterable(5)
Out[31]: False
```

在编写接受多种类型输入的函数时，我总是使用这个功能。常见的案例就是写接受任意序列类型（列表、元组、n维数组），甚至是一个迭代器的函数时使用这项功能。你可以先检查对象是否是一个列表（或者一个NumPy数组），如果不是就把它转换为列表：

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

2.3.1.9 导入

在Python中，模块就是以.py为后缀名并包含Python代码的文件。假设我们有以下模块：

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

假如我们想从另一个相同路径下的文件连接到some_module.py中定义的变量和函数，我们可以这样做：

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

或者：

```
from some_module import f, g, PI
result = g(5, PI)
```

通过使用as关键字，你可以对导入内容给予不同的变量名：

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

2.3.1.10 二元运算符和比较运算

大部分二元数学运算操作和比较运算都和你预想的一致：

```
In [32]: 5 - 7
Out[32]: -2

In [33]: 12 + 21.5
Out[33]: 33.5

In [34]: 5 <= 2
Out[34]: False
```

表2-3是所有可用的二元运算符。

检查两个引用是否指向同一个对象，可以使用is关键字。is not在你想检查两个对象不是相同对象时也是有效的。

```
In [35]: a = [1, 2, 3]

In [36]: b = a

In [37]: c = list(a)

In [38]: a is b
Out[38]: True

In [39]: a is not c
Out[39]: True
```

因为list函数总是创建一个新的Python列表（即一份拷贝），我们可以确定c与a是不同的。is和==是不同的，因为在这种情况下我们可以得到：

```
In [40]: a == c
Out[40]: True
```

is和is not的常用之处是检查一个变量是否为None，因为None只有一个实例：

```
In [41]: a = None

In [42]: a is None
Out[42]: True
```

表2-3：二元操作符

操作符	描述
a+b	a 加 b
a - b	a 减 b
a * b	a 乘 b
a / b	a 除以 b
a // b	a 整除以 b
a ** b	a 的 b 次方
a & b	a 与 b；对于整数则是按位 AND
a | b	a 或 b；对于整数则是按位 OR
a ^ b	对布尔值，a 异或 b；对整数则是按位异或
a == b	a 和 b 相等则为 True
a != b	a 和 b 不相等则为 True
a <= b, a < b	小于等于，小于
a > b, a >= b	大于，大于等于
a is b	a 和 b 是同一个 Python 对象则为 True
a is not b	a 和 b 不是同一个 Python 对象则为 True

2.3.1.11　可变对象和不可变对象

Python中的大部分对象，例如列表、字典、NumPy数组都是可变对象，大多数用户定义的类型（类）也是可变的。可变对象中包含的对象和值是可以被修改的：

```
In [43]: a_list = ['foo', 2, [4, 5]]

In [44]: a_list[2] = (3, 4)
```

```
In [45]: a_list
Out[45]: ['foo', 2, (3, 4)]
```

还有其他一些对象是不可变的，比如字符串、元组：

```
In [46]: a_tuple = (3, 5, (4, 5))

In [47]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

请牢记，你可以修改一个对象不代表你应该那么做。修改行为通常会有副作用。例如，当编写一个函数时，任何副作用都应当显式地在函数的文档或注释中告诉使用者。如果可能的话，我建议使用不可变性，避免副作用，尽管不可变对象中也可能包含可变对象。

2.3.2 标量类型

Python的标准库中拥有一个小的内建类型集合，用来处理数值数据、字符串、布尔值（True或False）以及日期和时间。这类的“单值”类型有时被称为标量类型，我们在本书中称之为标量。表2-4是主要的标量类型。由于标准库中含有datetime模块，日期和时间的处理将单独讨论。

表2-4：标准Python标量类型

类型	描述
None	Python 的 "null" 值（只存在一个实例）
str	字符串类型，包含 Unicode（UTF-8 编码）字符串
bytes	原生 ASCII 字节（或者 Unicode 编码字节）
float	双精度 64 位浮点数值（请注意并没有独立的 double 类型）
bool	True 或 False
int	任意精度无符号整数

2.3.2.1 数值类型

基础的Python数字类型就是int和float。int可以存储任意大小数字：

```
In [48]: ival = 17239871

In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

浮点数在Python中用float表示，每一个浮点数都是双精度64位数值。它们可以用科学计数法表示：

```
In [50]: fval = 7.243

In [51]: fval2 = 6.78e-5
```

整数除法会把结果自动转型为浮点数：

```
In [52]: 3 / 2
Out[52]: 1.5
```

如果需要C风格的整数除法（去除了非整数结果的小数部分），可以使用整除操作符//：

```
In [53]: 3 // 2
Out[53]: 1
```

2.3.2.2　字符串

很多人是因为Python强大、灵活的内建字符串操作功能而使用Python的。你可以用单引号'或双引号"写一个字符串字面值：

```
a = 'one way of writing a string'
b = "another way"
```

对于含有换行的多行字符串，你可以使用三个单引号'''或三个双引号"""：

```
c = """
This is a longer string that
spans multiple lines
"""
```

你可能会惊讶字符串c实际上包含了四行文本；"""后的换行和lines后的换行都是包含在字符串中的。我们可以使用count方法来计算c的回车符：

```
In [55]: c.count('\n')
Out[55]: 3
```

Python的字符串是不可变的，你无法修改一个字符串：

```
In [56]: a = 'this is a string'

In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment
```

```
In [58]: b = a.replace('string', 'longer string')

In [59]: b
Out[59]: 'this is a longer string'
```

操作后，变量a就是不可修改的：

```
In [60]: a
Out[60]: 'this is a string'
```

很多Python对象可以通过str函数转成字符串：

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```

字符串是Unicode字符的序列，因此可以被看作是除了列表和元组（下一章将会详细讨论）外的另一种序列：

```
In [64]: s = 'python'

In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'
```

s[：3]这种语法被称为切片，在多种Python序列中都有实现。本书后续将会更为详细地介绍，这个语法将在本书中广泛使用。

反斜杠符号\是一种转义符号，它用来指明特殊符号，比如换行符号\n或Unicode字符。如果要在字符串中写反斜杠则需要将其转义：

```
In [67]: s = '12\\34'

In [68]: print(s)
12\34
```

如果你有一个不含特殊符号但含有大量反斜杠的字符串时，你会觉得很烦。幸运的是，你可以在字符串前面加一个前缀符号r，表明这些字符是原生字符：

```
In [69]: s = r'this\has\no\special\characters'

In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

r是raw的简写，表示原生的。

将两个字符串结合到一起产生一个新字符串：

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```

字符串格式化是另一个重要主题。Python 3中进行字符串格式化的方式又拓展了。我将简明地介绍主流方式中的一种。字符串对象拥有一个format方法，可以用来代替字符串中的格式化参数，并产生一个新的字符串：

```
In [74]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

在这个字符串中，

·{0：.2f}表示将第一个参数格式化为2位小数的浮点数

·{1：s}表示将第二个参数格式化为字符串

·{2：d}表示将第三个参数格式化整数

为了替代这些格式化参数，我们将含有参数的序列传给format方法：

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1)
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

字符串格式化是一个深奥的主题；有多种方式和大量的选项可用于控制如何把值格式化到结果字符串中。如果要深入地了解这些，我建议参考Python官方文档（https://docs.python.org/3.6/library/string.html ）。

我将在第8章中更为详细地介绍如何在数据分析中进行通用字符串处理。

2.3.2.3 字节与Unicode

在现代Python中（此处指Python 3.0及以上），Unicode成为字符串类型的一等类，用于更好地兼容处理ASCII和非ASCII文本。在Python的早期版本中，字符串完全是字节，而没有显式的Unicode编码。如果你知道字符的编码，你就可以将其转换为Unicode。让我们看以下示例：

```
In [76]: val = "español"
In [77]: val
Out[77]: 'español'
```

我们可以使用enocde方法将这个Unicode字符串转换为UTF-8字节：

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

假设你知道一个字节对象的Unicode编码，你可以再使用decode方法进行解码：

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

由于偏好使用UTF-8编码的人越来越多，你会因为历史遗留问题在数据中碰到一些不同的编码：

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'
```

```
In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

在文件的上下文中，最常遇到的就是bytes（字节）对象，在字节对象中我们并不想让所有的数据都隐式地解码为Unicode字符串。

你可以在字符串前加前缀b来定义字符文本，尽管可能很少需要这么做：

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded # this is str (Unicode) now
Out[88]: 'this is bytes'
```

2.3.2.4 布尔值

Python中的布尔值写作True和False。比较运算和其他条件表达式的结果为True或False。布尔值可以与and和or关键字合用：

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

2.3.2.5 类型转换

str、bool、int和float既是数据类型，同时也是可以将其他数据转换为这些类型的函数：

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
```

```
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```

2.3.2.6 None

None是Python的null值类型。如果一个函数没有显式地返回值，则它会隐式地返回None：

```
In [97]: a = None

In [98]: a is None
Out[98]: True

In [99]: b = 5

In [100]: b is not None
Out[100]: True
```

None还可以作为一个常用的函数参数默认值：

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

从技术角度来说，None不仅是一个关键字，它还是NoneType类型的唯一实例：

```
In [101]: type(None)
Out[101]: NoneType
```

2.3.2.7 日期和时间

Python中内建的datetime模块，提供了datetime、data和time类型。可能正如你想象的，datetime类型是包含日期和时间信息的，最常用的方法是：

```
In [102]: from datetime import datetime, date, time

In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

对于datetime实例，你可以分别使用date和time方法获取它的date和time对象：

```
In [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)

In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

strftime方法将datetime转换为字符串

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')
Out[108]: '10/29/2011 20:30'
```

字符串可以通过strptime函数转换为datetime对象：

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```

表2-5是完整的格式化详细说明。

当你在聚合或分组时间序列数据时，会常常用到替代datetime时间序列中的一些值，比如将分钟、秒替换为0：

```
In [110]: dt.replace(minute=0, second=0)
Out[110]: datetime.datetime(2011, 10, 29, 20, 0)
```

由于datetime.datetime是不可变类型，以上的方法都是产生新的对象。

两个不同的datetime对象会产生一个datatime.timedelta类型的对象：

```
In [111]: dt2 = datetime(2011, 11, 15, 22, 30)

In [112]: delta = dt2 - dt

In [113]: delta
Out[113]: datetime.timedelta(17, 7179)

In [114]: type(delta)
Out[114]: datetime.timedelta
```

输出的timedelta（17，7179）表示时间间隔为17天又7，179秒。

将timedelta加到一个datetime上将产生一个新的对象。

```
In [115]: dt
Out[115]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [116]: dt + delta
Out[116]: datetime.datetime(2011, 11, 15, 22, 30)
```

表2-5：Datetime格式化详细说明（ISO C89兼容）

类型	描述
%Y	四位的年份
%y	两位的年份

表2-5：Datetime格式化详细说明（ISO C89兼容）（续）

类型	描述
%m	两位的月份 [01, 12]
%d	两份的天数值 [01, 31]
%H	小时值（24 小时制）[00, 23]
%I	小时值（12 小时制）[01, 12]
%M	两位的分钟值 [00，59]
%S	秒值 [00, 61]（60、61 用于区分闰秒）
%w	星期值 [0（星期天），6]
%U	一年中第几个星期的值 [00, 53]；星期天是每周第一天，第一个星期天前的一周是第 0 个星期
%W	一年中第几个星期的值 [00, 53]；星期一是每周第一天，第一个星期一前的一周是第 0 个星期
%z	UTC 时区偏置，格式为 +HHMM 或 -HHMM；如果是简单时区则为空
%F	%Y-%m-%d 的简写（例如 2012-4-18）
%D	%m/%d/%y 的简写（例如 04/18/12）

2.3.3 控制流

其他语言中的标准控制流概念在Python中也是通过一些条件逻辑、循环等内建关键字的方式实现的。

2.3.3.1 if、elif和else

if语句是最广为人知的控制流类型。它检查一个条件是否为True，请看以下代码示例：

```
if x < 0:
    print('It's negative')
```

一个if语句可以接多个elif代码块和一个else代码块，如果所有的elif条件均为False，则执行else代码块：

```
if x < 0:
    print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

如果某个条件为True，则后面的elif和else代码块则不会执行。当使用and和or进行混合条件判断时，条件判断是从左到右的并且在and或or两侧的条件会有“短路”现象：

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....:     print('Made it')
Made it
```

在本例中，c＞d是不会去判断的，因为第一个比较判断的值是True。

链式比较也是可以的：

```
In [120]: 4 > 3 > 2 > 1
Out[120]: True
```

2.3.3.2 for循环

for循环用于遍历一个集合（例如列表或元组）或一个迭代器。标准的for循环语法如下：

```
for value in collection:
    # 用值做些什么
```

使用continue关键字可以跳过conitnue后面的代码进入下一次循环。考虑以下代码，对列表中的非None值进行累加：

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

使用break关键字可以结束一个for循环。以下代码会对列表元素累加，直到5出现：

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

break关键字只结束最内层的for循环；外层的for循环会继续运行：

```
In [121]: for i in range(4):
   .....:     for j in range(4):
   .....:         if j > i:
   .....:             break
   .....:         print((i, j))
   .....:
(0, 0)
```

```
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

更详细地说，如果集合或迭代器中的元素是一个序列（比如元组或列表），它们可以在for循环语句中很方便地通过拆包成为变量：

```
for a, b, c in iterator:
    # 做些什么
```

2.3.3.3 while循环

while循环会在条件符合时一直执行代码块，直到条件判断为False或显式地以break结尾时才结束：

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

2.3.3.4 pass

pass就是Python中的“什么都不做”的语句。它用于在代码段中表示不执行任何操作（或者是作为还没有实现的代码占位符）；之所以需要它，是因为Python使用了缩进来分隔代码块：

```
if x < 0:
    print('negative!')
elif x == 0:
    # 在这里放点聪明的代码
    pass
else:
    print('positive!')
```

2.3.3.5 range

range函数返回一个迭代器，该迭代器生成一个等差整数序列：

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

起始、结尾、步进（可以是负的）可以传参给range函数：

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

如你所见，range产生的整数包含起始但不包含结尾。range常用于根据序列的索引遍历序列：

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

尽管你可以使用函数，比如list函数将range产生的整数存储在其他数据结构，但通常默认的迭代器形式就是你想要的。以下代码会将0到99，999中所有可以被3或5整除的整数相加：

```
sum = 0
for i in range(100000):
    # % 是求模操作符
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

虽然range产生的序列可以是任意大小，但在任意给定时间内的内存使用量是非常小的。

2.3.3.6 三元表达式

Python中的三元表达式允许你将一个if-else代码块联合起来，在一行代码或一个语句中生成数据。语法如下：

```
value = true-expr if condition else false-expr
```

此处的true-expr和false-expr可以是任意Python表达式.它与以下更详细的代码效果一致：

```
if condition:
    value = true-expr
else:
    value = false-expr
```

以下是具体的示例：

```
In [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: 'Non-negative'
```

在if-else代码块中，是按顺序逐个执行的。因此，三元表达式的“if”侧和“else”侧可能会包含计算消耗，但是只有真分支会被采用。

虽然我们可以使用三元表达式来压缩代码量，但请注意如果条件以及真假表达式非常复杂，可能会牺牲可读性。

第3章　内建数据结构、函数及文件

本章将讨论贯穿本书所要使用的Python语言内建功能。由于像pandas和NumPy这类附加库提供了在大数据集上的高级计算功能，所以它们被设计为与Python内建数据操作工具协同使用。

我们将开始介绍Python的常用数据结构：元组、列表、字典和集合。然后我们会讨论如何创建可复用的Python函数。我们将介绍Python文件对象的机制以及如何与你的本地硬盘进行交互。

3.1 数据结构和序列

Python的数据结构简单但强大。精通这些数据结构是成为优秀Python编程者的必要条件。

3.1.1　元组

元组是一种固定长度、不可变的Python对象序列。创建元组最简单的办法就是用逗号分隔序列值。

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```

当你通过更复杂的表达式来定义元组时，通常需要用括号将值包起来，例如下面这个例子。生成了元素是元组的元组：

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

你可以使用tuple函数将任意序列或迭代器转换为元组：

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

元组的元素可以通过中括号[]来获取，在大多数序列类型中都可以使用这个方法。和C、C++、Java以及很多其他语言一样，Python中的序列索引是从0开始的：

```
In [8]: tup[0]
Out[8]: 's'
```

虽然对象元组中存储的对象其自身是可变的，但是元组一旦创建，各个位置上的对象是无法被修改的：

```
In [9]: tup = tuple(['foo', [1, 2], True])
In [10]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

如果元组中的一个对象是可变的，例如列表，你可以在它内部进行修改：

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

可以使用+号连接元组来生成更长的元组：

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

将元组乘以整数，则会和列表一样，生成含有多份拷贝的元组：

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

请注意对象自身并没有复制，只是指向它们的引用进行了复制。

3.1.1.1 元组拆包

如果你想要将元组型的表达式赋值给变量，Python会对等号右边的值进行拆包：

```
In [15]: tup = (4, 5, 6)
In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

即使是嵌套元组也可以拆包：

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

使用这个功能，你可以轻易地交换变量名。在其他语言中，代码可能如下：

```
tmp = a
a = b
b = tmp
```

但在Python中，交换可以如下完成：

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

拆包的一个常用场景就是遍历元组或列表组成的序列：

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
   ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

另一个常用场景是从函数返回多个值。我将在后续内容详细介绍。

Python语言新增了一些更为高级的元组拆包功能，用于帮助你从元组的起始位置“采集“一些元素。这个功能使用特殊的语法*rest，用于在函数调用时获取任意长度的位置参数列表：

```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

rest部分有时是你想要丢弃的数据，rest这个变量名并没有什么特殊之处，为了方便，很多Python编程者会使用下划线（_）来表示不想要的变量：

```
In [33]: a, b, *_ = values
```

3.1.1.2　元组方法

由于元组的内容和长度是无法改变的，它的实例方法很少。一个常见的有用方法是count（列表中也可用），用于计量某个数值在元组中出现的次数：

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

3.1.2 列表

与元组不同，列表的长度是可变的，它所包含的内容也是可以修改的。你可以使用中括号[]或者list类型函数来定义列表：

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

列表与元组非常相似（尽管元组不可修改），它们的很多函数用法是相似的。

list函数在数据处理中常用于将迭代器或者生成器转化为列表：

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.1.2.1 增加和移除元素

使用append方法可以将元素添加到列表的尾部：

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

使用insert方法可以将元素插入到指定的列表位置：

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

插入位置的范围在0到列表长度之间。

 insert与append相比，计算代价更高。因为子序列元素不得不在内部移动为新元素提供空间。如果你想要在序列的头部和尾部都插入元素，那你应该探索下collections.deque，它是一个双端队列，可以满足头尾部都增加的要求。

insert的反操作是pop，该操作会将特定位置的元素移除并返回：

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

元素可以通过remove方法移除，该方法会定位第一个符合要求的值并移除它：

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

如果不考虑性能，通过使用append和remove，你可以将Python的列表用作一种完全合适的“多集合”数据结构。

使用in关键字可以检查一个值是否在列表中：

```
In [55]: 'dwarf' in b_list
```

```
Out[55]: True
```

not关键字可以用作in的反义词，表示“不在”：

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

与字典、集合（后面会介绍）相比，检查列表中是否包含一个值是非常缓慢的。这是因为Python在列表中进行了线性逐个扫描，而在字典和集合中Python是同时检查所有元素的（基于哈希表）。

3.1.2.2　连接和联合列表

与元组类似，两个列表可以使用+号连接：

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

如果你有一个已经定义的列表，你可以用extend方法向该列表添加多个元素：

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

请注意通过添加内容来连接列表是一种相对高代价的操作，这是因为连接过程中创建了新列表，并且还要复制对象。使用extend将元素添加到已经存在的列表是更好的方式，尤其是在你需要构建一个大型列表时：

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

上述实现比下述实现更快：

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

3.1.2.3 排序

你可以调用列表的sort方法对列表进行内部排序（无须新建一个对象）：

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```

sort有一些选项偶尔会派上用场。其中一项是传递一个二级排序key——一个用于生成排序值的函数。例如，我们可以通过字符串的长度进行排序：

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

下面，我们会讨论sorted方法，该方法可以针对通用序列产生一个排序后的拷贝。

3.1.2.4 二分搜索和已排序列表的维护

内建的bisect模块实现了二分搜索和已排序列表的插值。bisect.bisect会找到元素应当被插入的位置，并保持序列排序，而bisect.insort将元素插入到相应位置：

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4
```

```
In [70]: bisect.bisect(c, 5)
Out[70]: 6

In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

bisect模块的函数并不会检查列表是否已经排序，因为这么做代价太大。因此，对未排序列表使用bisect的函数虽然不会报错，但可能会导致不正确的结果。

3.1.2.5 切片

使用切片符号可以对大多数序列类型选取其子集，它的基本形式是将start：stop传入到索引符号[]中：

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

切片还可以将序列赋值给变量：

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

由于起始位置start的索引是包含的，而结束位置stop的索引并不包含，因此元素的数量是stop-start。

start和stop是可以省略的，如果省略的话会默认传入序列的起始位置或结束位置：

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

负索引可以从序列的尾部进行索引：

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

如果你以前使用过R或MATLAB，切片语义则需要适应一下。图3-1有助于理解通过正数或负数进行切片。在图中，索引值被显示在“箱体边缘”，有助于展示使用正数或负数进行切片选择的起始位置和结束位置。

步进值step可以在第二个冒号后面使用，意思是每隔多少个数取一个值：

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]
```

当需要对列表或元组进行翻转时，一种很聪明的用法就是向步进传值-1：

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

3.1.3 内建序列函数

Python有很多有用的序列函数，你应当熟悉并择机使用。

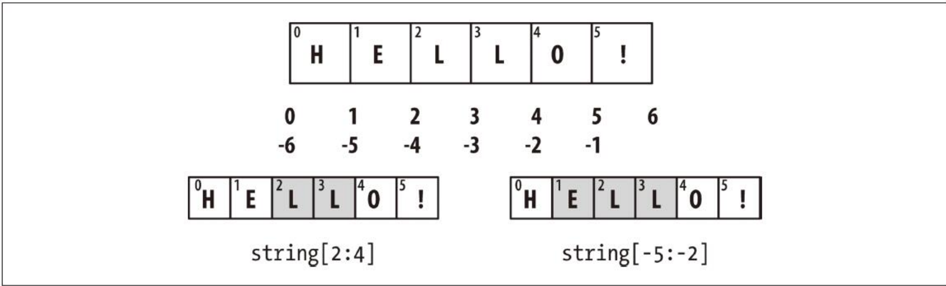


图3-1：Python切片示例

3.1.3.1 enumerate

我们经常需要在遍历一个序列的同时追踪当前元素的索引。一种自行实现的方法像下面的示例：

```
i = 0
for value in collection:
    # 使用值做点事
    i += 1
```

由于这种场景很常见，所以Python内建了enumerate函数，返回了（i，value）元组的序列，其中value是元素的值，i是元素的索引：

```
for i, value in enumerate(collection):
    # 使用值做点事
```

当你需要对数据建立索引时，一种有效的模式就是使用enumerate构造一个字典，将序列值（假设是唯一的）映射到索引位置上：

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
```

```
   ....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
3.1.3.2 sorted
```

3.1.3.2 sorted

sorted函数返回一个根据任意序列中的元素新建的已排序列表：

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

sorted函数接受的参数与列表的sort方法一致。

3.1.3.3 zip

zip将列表、元组或其他序列的元素配对，新建一个元组构成的列表：

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip可以处理任意长度的序列，它生成列表长度由最短的序列决定：

```
In [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

zip的常用场景为同时遍历多个序列，有时候会和enumerate同时使用：

```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

给定一个已“配对”的序列时，zip函数有一种机智的方式去“拆分”序列。这种方式的另一种思路就是将行的列表转换为列的列表。语法看上去略显魔幻：

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   ....:             ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

3.1.3.4 reversed

reversed函数将序列的元素倒序排列：

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

请牢记，reversed是一个生成器（将在后续内容讨论），因此如果没有实例化（例如使用list函数或进行for循环）的时候，它并不会产生一个倒序的列表。

3.1.4 字典

dict（字典）可能是Python内建数据结构中最重要的。它更为常用的名字是哈希表或者是关联数组。字典是拥有灵活尺寸的键值对集合，其中键和值都是Python对象。用大括号{}是创建字典的一种方式，在字典中用逗号将键值对分隔：

```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

你可以访问、插入或设置字典中的元素，就像访问列表和元组中的元素一样：

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

你可以用检查列表或元组中是否含有一个元素的相同语法来检查字典是否含有一个键：

```
In [107]: 'b' in d1
Out[107]: True
```

你可以使用del关键字或pop方法删除值，pop方法会在删除的同时返回被删的值，并删除键：

```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:

{'a': 'some value',
```

```
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}

In [110]: d1['dummy'] = 'another value'

In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}

In [112]: del d1[5]

In [113]: d1
Out[113]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

keys方法和values方法会分别为你提供字典键、值的迭代器。然而键值对并没有特定的顺序，这些函数输出的键、值都是按照相同的顺序：

```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

你可以使用update方法将两个字典合并：

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

update方法改变了字典中元素位置，因此对于任何原字典中已经存在的键，如果传给update方法的数据也含有相同的键，则它的值将会被覆盖。

3.1.4.1　从序列生成字典

通常情况下，你会有两个序列想要在字典中按元素配对。起初，你可能会写出这样的代码：

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

由于字典本质上是2-元组（含有2个元素的元组）的集合，字典是可以接受一个2-元组的列表作为参数的：

```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

稍后我们将会讨论字典推导式，那是另一种构建字典的方法。

3.1.4.2　默认值

通常情况下，会有这样的代码逻辑：

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

不过字典的get方法和pop方法可以返回一个默认值，因此上述的if-else代码块可以被简写为：

```
value = some_dict.get(key, default_value)
```

带有默认值的get方法会在key参数不是字典的键时返回None，而pop会抛出异常。一个常见的场景是字典中的值集合通过设置，成为另一种集

合，比如列表。举个例子，你可以想象一下将字词组成的列表根据首字母分类为包含列表的字典：

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [124]: by_letter = {}

In [125]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:

In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

字典的setdefault方法就是为了这个目的而产生的。上述的for循环语句可以被写为：

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

内建的集合模块有一个非常有用的类，defaultdict。这个类使得上述目的实现更为简单。想要生成符合要求的字典，你可以向字典中传入类型或能在各位置生成默认值的函数：

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

3.1.4.3 有效的字典键类型

尽管字典的值可以是任何Python对象，但键必须是不可变的对象，比如标量类型（整数、浮点数、字符串）或元组（且元组内对象也必须是不可变对象）。这里要使用到一个术语叫作哈希化，通过hash函数可以检查一个对象是否可以哈希化（即是否可以用作字典的键）：

```
In [127]: hash('string')
Out[127]: 5023931463650008331

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # 会因为列表是可变的失败
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # 会因为列表是可变的失败
TypeError: unhashable type: 'list'
```

为了将列表作为键，一种方式就是将其转换为元组，而元组只要它内部元素都可以哈希化，则它自己也可哈希化：

```
In [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(1, 2, 3): 5}
```

3.1.5 集合

集合是一种无序且元素唯一的容器。你可以认为集合也像字典，但是只有键没有值。集合可以有两种创建方式：通过set函数或者是用字面值集与大括号的语法：

```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```

集合支持数学上的集合操作，例如联合、交集、差集、对称差集。考虑以下示例集合：

```
In [135]: a = {1, 2, 3, 4, 5}
In [136]: b = {3, 4, 5, 6, 7, 8}
```

两个集合的联合就是两个集合中不同元素的并集。可以通过union方法或|二元操作符完成：

```
In [137]: a.union(b)
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}

In [138]: a | b
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}
```

交集包含了两个集合中同时包含的元素。可以使用&操作符或intersection方法获得交集：

```
In [139]: a.intersection(b)
Out[139]: {3, 4, 5}

In [140]: a & b
Out[140]: {3, 4, 5}
```

表3-1是常用的集合方法列表。

表3-1：Python集合操作

函数	替代方法	描述
a.add(x)	N/A	将元素 x 加入集合 a
a.clear()	N/A	将集合重置为空，清空所有元素
a.remove(x)	N/A	从集合 a 移除某个元素
a.pop()	N/A	移除任意元素，如果集合是空的抛出 keyError
a.union(b)	a|b	a 和 b 中的所有不同元素
a.update(b)	a|=b	将 a 的内容设置为 a 和 b 的并集
a.intersection(b)	a&b	a、b 中同时包含的元素
a.intersection_update(b)	a&= b	将 a 的内容设置为 a 和 b 的交集
a.difference(b)	a-b	在 a 不在 b 的元素
a.difference_update(b)	a-=b	将 a 的内容设为在 a 不在 b 的元素
a.symmetric_difference(b)	a^b	所有在 a 或 b 中，但不是同时在 a、b 中的元素
a.symmetric_difference_update(b)	a^=b	将 a 的内容设为所有在 a 或 b 中，但不是同时在 a、b 中的元素
a.issubset(b)	N/A	如果 a 包含于 b 返回 True

表3-1：Python集合操作（续）

函数	替代方法	描述
a.issuperset(b)	N/A	如果 a 包含 b 返回 True
a.isdisjoint(b)	N/A	a、b 没有交集返回 True

所有的逻辑集合运算都有对应操作，允许你用操作的结果替代操作左边的集合内容。对于大型集合，下面的代码效率更高：

```
In [141]: c = a.copy()

In [142]: c |= b

In [143]: c
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}

In [144]: d = a.copy()

In [145]: d &= b
```

```
In [146]: d
Out[146]: {3, 4, 5}
```

和字典类似，集合的元素必须是不可变的。如果想要包含列表型的元素，必须先转换为元组：

```
In [147]: my_data = [1, 2, 3, 4]

In [148]: my_set = {tuple(my_data)}

In [149]: my_set
Out[149]: {(1, 2, 3, 4)}
```

你还可以检查一个集合是否是另一个结合的子集（包含于）或超集（包含）：

```
In [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

当且仅当两个集合的内容一模一样时，两个集合才相等：

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: True
```

3.1.6 列表、集合和字典的推导式

列表推导式是最受欢迎的Python语言特性之一。它允许你过滤一个容器的元素，用一种简明的表达式转换传递给过滤器的元素，从而生成一个新的列表。列表推导式的基本形式为：

```
[expr for val in collection if condition]
```

这与下面的for循环是等价的：

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

过滤条件是可以忽略的，只保留表达式。例如，给定一个字符串列表，我们可以过滤出长度大于2的，并且将字母改为大写：

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

集合与字典的推导式是列表推导式的自然拓展，用相似的方式生成集合与字典。字典推导式如下所示：

```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```

集合推导式看起来很像列表推导式，只是中括号变成了大括号：

```
set_comp = {expr for value in collection if condition}
```

和列表推导式类似，集合、字典的推导式非常方便，它们使代码更易读易写。如果有一个字符串的列表，假设我们想要一个集合，集合里包含

列表中字符串的长度，我可以通过集合推导式很方便地实现：

```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

我们也可以使用map函数更函数化、更简洁地表达：

```
In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

我们创建一个将字符串与其位置相匹配的字典作为字典推导式的简单示例：

```
In [159]: loc_mapping = {val : index for index, val in enumerate(str

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python':
```

3.1.6.1 嵌套列表推导式

假设我们有一个包含列表的列表，内容是英语姓名和西班牙语姓名：

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
   .....:             ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar'
```

你都已经忘了这些名字来自哪些文件，但想要根据语言来组织这些名字。现在再假设我们想要获得一个列表包含所有含有2个以上字母e的名字，我们当然可以简单地使用for循环：

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

实际上，你可以将整个操作用一个嵌套列表推导式来完成，如下：

```
In [162]: result = [name for names in all_data for name in names
   .....:           if name.count('e') >= 2]

In [163]: result
Out[163]: ['Steven']
```

首先，嵌套列表推导式可能会让你有点晕头转向。列表推导式的for循环部分是根据嵌套的顺序排列的，所有的过滤条件像之前一样被放在尾部。下面的例子是将含有整数元组的列表扁平化为一个简单的整数列表：

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [165]: flattened = [x for tup in some_tuples for x in tup]

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

请牢记for表达式的顺序应当和你写嵌套for循环来替代列表推导式的顺序一致：

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

你当然可以嵌套多层的列表推导式，但超过两到三层之后你很可能开始疑惑这种做法是否会有利于代码可读性。嵌套推导式的语法要和列表推导式中的列表推导式区分开，列表推导式中的列表推导式也是非常有效的：

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

上面的代码创建了一个包含列表的列表，而不是所有内部元素扁平排列。

3.2 函数

函数是Python中最重要、最基础的代码组织和代码复用方式。根据经验，如果你需要多次重复相同或类似的代码，就非常值得写一个可复用的函数。通过给一组Python语句一个函数名，形成的函数可以使你的代码更加可读。

函数声明时使用def关键字，返回时使用return关键字：

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

有多条返回语句是没有问题的。如果Python达到函数的尾部时仍然没有遇到return语句，就会自动返回None。

每个函数都可以有位置参数和关键字参数。关键字参数最常用于指定默认值或可选参数。在前面的函数中，x和y是位置参数，z是关键字参数。这意味着函数可以通过以下任意一种方式进行调用：

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

函数参数的主要限制是关键字参数必须跟在位置参数（如果有的话）后。你可以按照任意顺序指定关键字参数，这可以让你不必强行记住函数参数的顺序，而只需用参数名指定。

也可以使用关键字参数向位置参数传参。在前面的例子中，我们也可以这样写：

```
my_function(x=5, y=6, z=7)
my_function(y=6, x=5, z=7)
```

在部分场景中，这样做有助于代码可读性。

3.2.1 命名空间、作用域和本地函数

函数有两种连接变量的方式：全局、本地。在Python中另一种更贴切地描述变量作用域的名称是命名空间。在函数内部，任意变量都是默认分配到本地命名空间的。本地命名空间是在函数被调用时生成的，并立即由函数的参数填充。当函数执行结束后，本地命名空间就会被销毁（除了一些本章范围外的特殊情况）。考虑以下函数：

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

当调用func（）时，空的列表a会被创建，五个元素被添加到列表，之后a会在函数退出时被销毁。假设我们像下面这样声明a：

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

在函数外部给变量赋值是可以的，但是那些变量必须使用global关键字声明为全局变量：

```
In [168]: a = None

In [169]: def bind_a_variable():
   .....:     global a
   .....:     a = []
   .....: bind_a_variable()
   .....:

In [170]: print(a)
[]
```


我简单地讲下global关键字的用法。通常全局变量用来存储系统中的某些状态。如果你发现你大量使用了全局变量，可能表明你需要面向对象编程（使用类）。

3.2.2 返回多个值

当我在使用Java和C++编程后第一次使用Python编程时，我最喜欢的特性就是使用简单语法就可以从函数中返回多个值，代码如下：

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

在数据分析和其他科研应用中，你可能会发现经常需要返回多个值。这里实质上是返回了一个对象，也就是元组，而元组之后又被拆包为多个结果变量。在前面的例子中，我们可以用下面的代码代替：

```
return_value = f()
```

在这个例子中，return_value是一个包含3个元素的元组。像之前那样一次返回多个值还有一种潜在的、更有吸引力的实现：

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

具体用哪种技术取决于你需要做什么。

3.2.3 函数是对象

由于Python的函数是对象，很多在其他语言中比较难的构造在Python中非常容易实现。假设我们正在做数据清洗，需要将一些变形应用到下列字符串列表中：

```
In [171]: states = ['   Alabama ', 'Georgia!', 'Georgia', 'georgia',
   .....:           'south   carolina##', 'West virginia?']
```

任何处理过用户提交数据的人都对这样的数据感到凌乱。为了使这些数据整齐、可用于分析，有很是事情需要做：去除空格，移除标点符号，调整适当的大小写。一种方式是使用内建的字符串方法，结合标准库中的正则表达式模块re：

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

结果如下：

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

另一种会让你觉得有用的实现就是将特定的列表操作应用到某个字符串的集合上：

```
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

结果如下：

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

像这种更为函数化的模式可以使你在更高层次上方便地修改字符串变换方法。clean_strings函数现在也具有更强的复用性和通用性。

你可以将函数作为一个参数传给其他的函数，比如内建的map函数，可以将一个函数应用到一个序列上：

```
In [176]: for x in map(remove_punctuation, states):
   .....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

3.2.4 匿名（Lambda）函数

Python支持所谓的匿名或lambda函数。匿名函数是一种通过单个语句生成函数的方式，其结果是返回值。匿名函数使用lambda关键字定义，该关键字仅表达“我们声明一个匿名函数”的意思：

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

本书的其余部分，我会更偏向使用匿名函数。你将会看到，匿名函数在数据分析中非常方便，因为在很多案例中数据变形函数都可以作为函数的参数。匿名函数代码量小（也更为清晰），将它作为参数进行传值，比写一个完整的函数或者将匿名函数赋值给局部变量更好。举个例子，考虑下面的不佳示例：

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

你也可以写成[x*2for x in ints]，但是在这里我们能够简单地将一个自定义操作符传递给apply_to_list函数。

另一个例子，假设你想要根据字符串中不同字母的数量对一个字符串集合进行排序：

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

这里我们可以将一个匿名函数传给列表的sort方法：

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

 和def关键字声明的函数不同，匿名函数对象自身并没有一个显式的__name__属性，这是lambda函数被称为匿名函数的一个原因。

3.2.5 柯里化：部分参数应用

柯里化是计算机科学术语（以数学家Haskell Curry命名），它表示通过部分参数应用的方式从已有的函数中衍生出新的函数。例如，假设我们有一个不重要的函数，其功能是将两个数加一起：

```
def add_numbers(x, y):
    return x + y
```

使用这个函数，我们可以衍生出一个只有一个变量的新函数，add_five，可以给参数加上5：

```
add_five = lambda y: add_numbers(5, y)
```

第二个参数对于函数add_numers就是柯里化了。这里并没有什么神奇的地方，我们真正做的事只是定义了一个新函数，这个新函数调用了已经存在的函数。内建的functools模块可以使用pratial函数简化这种处理：

```
from functools import partial
add_five = partial(add_numbers, 5)
```

3.2.6 生成器

通过一致的方式遍历序列，例如列表中的对象或者文件中的一行行内容，这是Python的一个重要特性。这个特性是通过迭代器协议来实现的，迭代器协议是一种令对象可遍历的通用方式。例如，遍历一个字典，获得字典的键：

```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [181]: for key in some_dict:
   .....:     print(key)
a
b
c
```

当你写下for key in some_dict的语句时，Python解释器首先尝试根据some_dict生成一个迭代器：

```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7fbbd5a9f908>
```

迭代器就是一种用于在上下文中（比如for循环）向Python解释器生成对象的对象。大部分以列表或列表型对象为参数的方法都可以接收任意的迭代器对象。包括内建方法比如min、max和sum，以及类型构造函数比如list和tuple：

```
In [184]: list(dict_iterator)
Out[184]: ['a', 'b', 'c']
```

生成器是构造新的可遍历对象的一种非常简洁的方式。普通函数执行并一次返回单个结果，而生成器则“惰性”地返回一个多结果序列，在每一个元素产生之后暂停，直到下一个请求。如需创建一个生成器，只需要在函数中将返回关键字return替换为yield关键字：

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
```

```
    for i in range(1, n + 1):
        yield i ** 2
```

当你实际调用生成器时，代码并不会立即执行：

```
In [186]: gen = squares()
In [187]: gen
Out[187]: <generator object squares at 0x7fbbd5ab4570>
```

直到你请求生成器中的元素时，它才会执行它的代码：

```
In [188]: for x in gen:
   .....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

3.2.6.1 生成器表达式

用生成器表达式来创建生成器更为简单。生成器表达式与列表、字典、集合的推导式很类似，创建一个生成器表达式，只需要将列表推导式的中括号替换为小括号即可：

```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7fbbd5ab29e8>
```

上面的代码与下面更为复杂的生成器是等价的：

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

在很多情况下，生成器表达式可以作为函数参数用于替代列表推导式：

```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350
```

```
In [192]: dict((i, i **2) for i in range(5))
Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

3.2.6.2 itertools模块

标准库中的itertools模块是适用于大多数数据算法的生成器集合。例如，groupby可以根据任意的序列和一个函数，通过函数的返回值对序列中连续的元素进行分组，参见下面的例子：

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven'

In [196]: for letter, names in itertools.groupby(names, first_letter
     .....:     print(letter, list(names)) # names is a generator
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

表3-2是一些我认为经常用到的itertools函数的列表。你可以通过查询Python官方文档来获得更多关于内建工具库的信息。

表3-2：一些有用的itertools函数

函数	描述
combinations (iterable, k)	根据 iterable 参数中的所有元素生成一个包含所有可能 K- 元组的序列，忽略元素的顺序，也不进行替代（需要替代请参考函数 combinations_with_replacement）
permutations (iterable, k)	根据 iterable 参数中的所有元素按顺序生成包含所有可能 K 元组的序列
groupby (iterable[, keyfunc])	根据每一个独一的 key 生成（key, sub-iterator）元组
product (*iterables, repeat=1)	以元组的形式，根据输入的可遍历对象生成笛卡尔积，与嵌套的 for 循环类似

3.2.7 错误和异常处理

优雅地处理Python的错误或异常是构建稳定程序的重要组成部分。在数据分析应用中，很多函数只能处理特定的输入。例如，Python的float函数可以将字符串转换为浮点数字，但是对不正确的输入会产生ValueError：

```
In [197]: float('1.2345')
Out[197]: 1.2345

In [198]: float('something')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-198-439904410854> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```

假设我们想要在float函数运行失败时可以优雅地返回输入参数。我们可以通过将float函数写入一个try/except代码段来实现：

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

如果float（x）执行时抛出了异常，则代码段中的except部分代码将会被执行：

```
In [200]: attempt_float('1.2345')
Out[200]: 1.2345

In [201]: attempt_float('something')
Out[201]: 'something'
```

你可能会注意到，除了ValueError，float函数还会抛出其他的异常：

```
In [202]: float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-202-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not 'tuple
```

你可能只想处理ValueError，因为TypeError（输入的不是字符串或数值）可能表明你的程序中有个合乎语法的错误。为了实现这个目的，在except后面写下异常类型：

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

然后我们可以得到：

```
In [204]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-204-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number, not 'tuple
```

你可以通过将多个异常类型写成元组的方式同时捕获多个异常（小括号是必不可少的）：

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

某些情况下，你可能想要处理一个异常，但是你希望一部分代码无论try代码块是否报错都要执行。为了实现这个目的，使用finally关键字：

```
f = open(path, 'w')
```

```
try:
    write_to_file(f)
finally:
    f.close()
```

这样，我们可以让f在程序结束后总是关闭。类似地，你可以使用else来执行当try代码块成功执行时才会执行的代码：

```
f = open(path, 'w')
try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

3.2.7.1 IPython中的异常

如果当你正在用%run执行一个脚本或执行任何语句时报错，IPython将会默认打印出完整的调用堆栈跟踪（报错追溯），会将堆栈中每个错误点附近的几行上下文代码打印出：

```
In [10]: %run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_thing
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exce
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:
```

比标准Python解释器提供更多额外的上下文是IPython的一大进步（标准Python解释器不提供任何额外的上下文）。你可以使用%xmode命令来控制上下文的数量，可以从Plain（普通）模式（与标准Python解释器一致）切换到Verbose（复杂）模式（可以显示函数的参数值以及更多有用信息）。你将会在下一章看到如何在错误发生后进入异常堆栈（使用%debug或%pdb命令）进行交互式事后调试。

3.3 文件与操作系统

本书的大部分内容使用了像pandas.read_csv这类的高级工具来从硬盘中将数据文件读取为Python数据结构。但是，理解如何在Python中处理文件的基础知识也是很重要的。幸运的是Python中处理文件非常简单，这也是Python能够在文本和文件处理领域如此流行的原因。

打开文件进行读取或写入，需要使用内建函数open和绝对、相对路径：

```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

默认情况下，文件是以只读模式'r'打开的。之后我们可以像处理列表一样处理文件f，并遍历f中的行内容，代码如下：

```
for line in f:
    pass
```

行内容会在行结尾标识（EOL）完整的情况下从文件中全部读出，所以你会经常看到一些代码，功能是将文件中的内容形成不带EOL的列表：

```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sue '']
```

当你使用open来创建文件对象时，在结束操作时显式地关闭文件是非常重要的。关闭文件会将资源释放回操作系统：

```
In [211]: f.close()
```

另一种更简单的关闭文件的方式就是使用with语句：

```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

使用with语句，文件会在with代码块结束后自动关闭。

如果我们写下f＝open（path，'w'），一个新的文件会在examples/segismundo.txt的位置被创建（请小心！），并在同一路径下覆盖同名文件。还有一种'x'文件模式，它会创建可写的文件，但如果给定路径下已经存在同名文件就会创建失败。参见表3-3中列出的全部有效文件读写模式。

对于可读文件，最常用的方法是read、seek和tell。read返回文件中一定量的字符，构成字符的内容是由文件的编码决定的（例如UTF-8），或者在二进制模式下打开文件读取简单的原生字节：

```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'SueOut[216]: b'Sue\xc3\xb1a el '
```

read方法通过读取的字节数来推进文件句柄的位置。tell方法可以给出句柄当前的位置：

```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

尽管我们从文件中读取了10个字符，但是当前的句柄位置是11，这是因为使用默认编码的情况下需要这么多字节来对10个字符进行解码。你可以用sys模块来检查文件的默认编码：

```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

seek方法可以将句柄位置改变到文件中特定的字节：

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: '
```

之后，请牢记关闭文件：

```
In [223]: f.close()

In [224]: f2.close()
```

表3-3：Python文件模式

模式	描述
r	只读模式
w	只写模式，创建新文件（清除路径下的同名文件中的数据）
x	只写模式，创建新文件，但存在同名路径时会创建失败
a	添加到已经存在的文件（如果不存在就创建）
r+	读写模式
b	二进制文件的模式，添加到别的模式中（比如 'rb' 或 'wb'）
t	文件的文本模式（自动将字节解码为 Unicode）。如果没有指明模式，默认使用此模式，可以添加到别的模式中（例如 'rt' 或 'xt'）

为了将本文写入文件，你可以使用文件对象的write或wirtelines方法。例如，我们可以创建一个没有空行的prof_mod.py，像这样：

```
In [225]: with open('tmp.txt', 'w') as handle:
   .....:     handle.writelines(x for x in open(path) if len(x) > 1)

In [226]: with open('tmp.txt') as f:
   .....:     lines = f.readlines()

In [227]: lines
Out[227]:
['Sue 'aunque ninguno lo entiende.\n']
```

表3-4是很多常用的文件方法。

表3-4：重要的Python文件方法或属性

方法	描述
read([size])	将文件数据作为字符串返回，可选参数 size 控制读取的字节数
readlines([size])	返回文件中行内容的列表，size 参数可选
write(str)	将字符串写入文件

表3-4：重要的Python文件方法或属性（续）

方法	描述
writelines(strings)	将字符串序列写入文件
close()	关闭文件
flush()	将内部 I/O 缓冲器内容刷新到硬盘
seek(pos)	移动到指定的位置（整数）
tell()	返回当前的文件位置，返回值是整数
closed	如果文件已关闭，则为 True

3.3.1 字节与Unicode文件

默认的Python文件行为（无论是可读或可写）是文本模式，这意味着你需要处理Python字符串（比如Unicode）。与二进制模式不同，在二进制模式中你可以将b添加到文件模式中，然后进行读写。我们来看下上一节中的文件（包含了UTF-8编码的非ASCII字符）：

```
In [230]: with open(path) as f:
   .....:     chars = f.read(10)

In [231]: chars
```

Out[231]：'SueUTF-8是一种可变长度的Unicode编码，因此当我从文件中请求一定量的字符时，Python从文件中读取了足够的字节（可以少至10个字节，也可以多至40个字节），并进行了解码。如果我使用'rb'模式代替，则read请求提取了一定量的字节：

```
In [232]: with open(path, 'rb') as f:
   .....:     data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

根据文本编码，你可能会将字节解码为str对象，但是只有每个已编码的Unicode字符是完整的情况下，才能进行解码：

```
In [234]: data.decode('utf8')
Out[234]: 'SueUnicodeDecodeError: 'utf-8' codec can't decode byte 0xc
d end of data
```

文本模式下，利用open方法的选项参数encoding，Python提供了一种方便的方法将文件内容从Unicode编码转换为其他类型的编码：

```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
   .....: with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
   .....:     sink.write(source.read())
```

```
In [238]: with open(sink_path, encoding='iso-8859-1') as f:
   .....:     print(f.read(10))
```

Sue除了二进制模式，在打开文件时使用seek要当心。如果文件的句柄位置恰好在一个Unicode符号的字节中间时，后续的读取会导致错误：

```
In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sue/miniconda/envs/book-env/lib/python3.6/codecs.py in dec
    319    # 对输入解码（考虑buffer）
    320    data = self.buffer + input
--> 321    (result, consumed) = self._buffer_decode(data, self.errors,
)
    322          # 保持未解码的输入直到下一次调用
    323          self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position
tart byte

In [244]: f.close()
```

如果你发现你常常需要在非ASCII文本数据上进行数据分析，那么精通Python的Unicode功能是很有必要的。参见Python官方文档（https://docs.python.org/ ）可以了解更多相关信息

3.4 本章小结

在掌握了Python环境和语言的基础后，现在是时候学习NumPy和Python中的面向数组计算了。

第4章　NumPy基础：数组与向量化计算

NumPy，是Numerical Python的简称，它是目前Python数值计算中最为重要的基础包。大多数计算包都提供了基于NumPy的科学函数功能，将NumPy的数组对象作为数据交换的通用语。

以下内容将会出现在NumPy中：

·ndarray，一种高效多维数组，提供了基于数组的便捷算术操作以及灵活的广播功能。

·对所有数据进行快速的矩阵计算，而无须编写循环程序。

·对硬盘中数组数据进行读写的工具，并对内存映射文件进行操作。

·线性代数、随机数生成以及傅里叶变换功能。

·用于连接NumPy到C、C＋＋和FORTRAN语言类库的C语言API。

由于NumPy提供了一个非常易用的C语言API，这使得，将数据传递给用底层语言编写的外部类库，再由外部类库将计算结果按照NumPy数组的方式返回，变得非常简单。这个特征使得Python可以对存量C/C＋＋/Fortran代码库进行封装，并为这些代码提供动态、易用的接口。

NumPy本身并不提供建模和科学函数，理解NumPy的数组以及基于数组的计算将帮助你更高效地使用基于数组的工具，比如pandas。由于NumPy是一个很大的话题，我将在后续章节讲解NumPy的一些高级特性，例如广播机制（见附录A）。

对于大多数的数据分析应用，我主要关注的内容为：

·在数据处理、清洗、构造子集、过滤、变换以及其他计算中进行快速的向量化计算。

·常见的数组算法，比sort、unique以及set操作等。

·高效的描述性统计和聚合/概述数据。

·数据排列和相关数据操作，例如对异构数据进行merge和join。

·使用数组表达式来表明条件逻辑，代替if-elif-else条件分支的循环。

·分组数据的操作（聚合、变换以及函数式操作）。

虽然NumPy提供了数值数据操作的计算基础，但大多数读者还是想把pandas作为统计、分析的基石，尤其是针对表格数据。pandas提供了更多的针对特定场景的函数功能，例如时间序列操作等NumPy并不包含的功能。

 Python中的数组计算方式要追溯到1995年，当时Jim Hugunin创造Numeric库。之后10年里，许多科研编程社区开始利用Python进行数组编程，但类库的生态在2000年之后都是碎片化的。2005年，Travis Oliphant在Numeric和Numarray项目之上打造了NumPy，将社区整合到同一个数组计算框架下。

NumPy之所以如此重要，其中一个原因就是它的设计对于含有大量数组的数据非常有效。此外还有如下原因：

·NumPy在内部将数据存储在连续的内存块上，这与其他的Python内建数据结构是不同的。NumPy的算法库是用C语言写的，所以在操作数据内存时，不需要任何类型检查或者其他管理操作。NumPy数组使用的内存量也小于其他Python内建序列。

·NumPy可以针对全量数组进行复杂计算而不需要写Python循环。

下面的例子将展现NumPy的不同，假设一个NumPy数组包含100万个整数，还有一个同样数据内容的Python列表：

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

现在我们同时对每个序列乘以2：

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

NumPy的方法比Python方法要快10到100倍，并且使用的内存也更少。

4.1 NumPy ndarray：多维数组对象

NumPy的核心特征之一就是N-维数组对象——ndarray。ndarray是Python中一个快速、灵活的大型数据集容器。数组允许你使用类似于标量的操作语法在整块数据上进行数学计算。

为了让你感受下NumPy如何使用类似于Python内建对象的标量计算语法进行批量计算，我首先导入NumPy，再生成一个小的随机数组：

```
In [12]: import numpy as np

# 生成随机数组
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

然后给data加上一个数学操作：

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894,  -5.1944],
       [ -5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

在第一个数学操作中，所有的元素都同时乘以了10。在第二个数学操作中，数组中的对应元素进行了相加。

本章及全书，我都使用标准的NumPy导入方式import numpy as np。你当然也可以在代码中写from numpy import*来省略多写的一个np.，然而我还是建议你保持写标准导入的方式。numpy这个命名空间包含了大量与Python内建函数重名的函数（比如min和max）

一个ndarray是一个通用的多维同类数据容器，也就是说，它包含的每一个元素均为相同类型。每一个数组都有一个shape属性，用来表征数

组每一维度的数量；每一个数组都有一个dtype属性，用来描述数组的数据类型：

```
In [17]: data.shape
Out[17]: (2, 3)
In [18]: data.dtype
Out[18]: dtype('float64')
```

本章将介绍NumPy数组的基础操作，这部分内容将足以保证本书后面内容的阅读。虽然深入理解NumPy对于大部分数据分析应用并不是必需的，但是精通基于数组的编程和思考是成为Python科学计算专家的重要一步。

 在本书中，当你看到“数组”、“NumPy数组”或“ndarray”时，他们都表示同一个对象：ndarray对象。

4.1.1 生成ndarray

生成数组最简单的方式就是使用array函数。array函数接收任意的序列型对象（当然也包括其他的数组），生成一个新的包含传递数据的NumPy数组。例如，列表的转换就是一个好例子：

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

嵌套序列，例如同等长度的列表，将会自动转换成多维数组：

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

因为data2是一个包含列表的列表，所以Numpy数组arr2形成了二维数组。我们可以通过检查ndim和shape属性来确认这一点：

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

除非显式地指定，否则np.array会自动推断生成数组的数据类型。数据类型被存储在一个特殊的元数据dtype中。例如，之前的两个例子：

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
```

```
Out[28]: dtype('int64')
```

除了np.array，还有很多其他函数可以创建新数组。例如，给定长度及形状后，zeros可以一次性创造全0数组，ones可以一次性创造全1数组。empty则可以创建一个没有初始化数值的数组。想要创建高维数组，则需要为shape传递一个元组：

```
In [29]: np.zeros(10)
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],
       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

想要使用np.empty来生成一个全0数组，并不安全，有些时候它可能会返回未初始化的垃圾数值。

arange是Python内建函数range的数组版：

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
```

表4-1展示的是标准数组的生成函数。由于NumPy专注于数值计算，如果没有特别指明的话，默认的数据类型是float64（浮点型）。

表4-1：数组生成函数

函数名	描述
array	将输入数据（可以是列表、元组、数组以及其他序列）转换为ndarray，如不显式指明数据类型，将自动推断；默认复制所有的输入数据
asarray	将输入转换为ndarray，但如果输入已经是ndarray则不再复制
arange	Python内建函数range的数组版，返回一个数组

表4-1：数组生成函数（续）

函数名	描述
ones	根据给定形状和数据类型生成全1数组
ones_like	根据所给的数组生成一个形状一样的全1数组
zeros	根据给定形状和数据类型生成全0数组
zeros_like	根据所给的数组生成一个形状一样的全0数组
empty	根据给定形状生成一个没有初始化数值的空数组
empty_like	根据所给数组生成一个形状一样但没有初始化数值的空数组
full	根据给定的形状和数据类型生成指定数值的数组
full_like	根据所给的数组生成一个形状一样但内容是指定数值的数组
eye, identity	生成一个N×N特征矩阵（对角线位置都是1，其余位置是0）

4.1.2　ndarray的数据类型

数据类型，即dytpe，是一个特殊的对象，它包含了ndarray需要为某一种类型数据所申明的内存块信息（也称为元数据，即表示数据的数据）：

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

dtype是NumPy能够与其他系统数据灵活交互的原因。通常，其他系统提供一个硬盘或内存与数据的对应关系，使得利用C或Fortran等底层语言读写数据变得十分方便。数据的dtype通常都是按照一个方式命名：类型名，比如float和int，后面再接上表明每个元素位数的数字。一个标准的双精度浮点值（Python中数据类型为float），将使用8字节或64位。因此，这个类型在NumPy中称为float64。表4-2将展现所有的NumPy所支持的数据类型。

 不要担心如何记住NumPy数据类型，尤其当你还是新手的时候。通常你只需要关心数据的大类，比如是否是浮点型、整数、布尔值、字符串或某个Python对象。当你需要在内存或硬盘上做更深入的存取操作时，尤其是大数据集时，你才真正需要了解存储的数据类型。

表4-2：NumPy数据类型

类型	类型代码	描述
int8, uint8	i1, u1	有符号和无符号的 8 数位整数
int16, uint16	i2, u2	有符号和无符号的 16 数位整数
int32, uint32	i4, u4	有符号和无符号的 32 数位整数
int64, uint64	i8, u8	有符号和无符号的 64 数位整数
float16	f2	半精度浮点数
float32	f4 或 f	标准单精度浮点数；兼容 C 语言 float
float64	f8 或 d	标准双精度浮点数；兼容 C 语言 double 和 Python float
float128	f16 或 g	拓展精度浮点数
complex64, complex128, complex256	c8,c16,c32	分别基于 32 位、64 位、128 位浮点数的复数
bool	?	布尔值，存储 True 或 False
object	O	Python object 类型
string_	S	修正的 ASCⅡ字符串类型；例如生成一个长度为 10 的字符串类型，使用 'S10'
unicode_	U	修正的 Unicode 类型，生成一个长度为 10 的 Unicode 类型，使用 'U10'

你可以使用astype方法显式地转换数组的数据类型：

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

在上面例子中，整数被转换成了浮点数。如果我把浮点数转换成整数，则小数点后的部分将被消除：

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])
```

```
In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

如果你有一个数组，里面的元素都是表达数字含义的字符串，也可以通过astype将字符串转换为数字：

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np

In [45]: numeric_strings.astype(float)
Out[45]: array([  1.25,  -9.6 ,  42.  ])
```

 在NumPy中，当使用numpy.string_类型作字符串数据要小心，因为NumPy会修正它的大小或删除输入且不发出警告。pandas在处理非数值数据时有更直观的开箱型操作。

如果因为某些原因导致转换类型失败（比如字符串无法转换为float64位时），将会抛出一个ValueError。这里我偷懒地使用float来代替np.float64，是因为NumPy可以使用相同别名来表征与Python精度相同的Python数据类型。

你也可以使用另一个数组的dtype属性：

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

也可以使用类型代码来传入数据类型：

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:

array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

 使用astype时总是生成一个新的数组，即使你传入的dtype与之前

一样。

4.1.3 NumPy数组算术

数组之所以重要是因为它允许你进行批量操作而无须任何for循环。NumPy用户称这种特性为向量化。任何在两个等尺寸数组之间的算术操作都应用了逐元素操作的方式：

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

带有标量计算的算术操作，会把计算参数传递给数组的每一个元素：

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

同尺寸数组之间的比较，会产生一个布尔值数组：

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])

In [59]: arr2 > arr
```

```
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

不同尺寸的数组间的操作，将会用到广播特性，将会在附录A中介绍。对于本书大部分内容来说，并不需要深入理解广播特性。

4.1.4 基础索引与切片

NumPy数组索引是一个大话题，有很多种方式可以让你选中数据的子集或某个单个元素。一维数组比较简单，看起来和Python的列表很类似：

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

如你所见，如果你传入了一个数值给数组的切片，例如arr[5：8]＝12，数值被传递给了整个切片。区别于Python的内建列表，数组的切片是原数组的视图。这意味着数据并不是被复制了，任何对于视图的修改都会反映到原数组上。

例子如下：

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

当我改变arr_slice，变化也会体现在原数组上：

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])
```

不写切片值的[：]将会引用数组的所有值：

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

假如你是NumPy新手，你可能会感到惊讶，因为其他的数组编程语言都是更为急切地复制数据。由于NumPy被设计成适合处理非常大的数组，你可以想象如果NumPy持续复制数据会引起多少内存问题。

 如果你还是想要一份数组切片的拷贝而不是一份视图的话，你就必须显式地复制这个数组，例如arr[5：8].copy（）

对更高维度的数组，你会有更多选择。在一个二维数组中，每个索引值对应的元素不再是一个值，而是一个一维数组：

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

因此，单个元素可以通过递归的方式获得。但是要多写点代码，你可以通过传递一个索引的逗号分隔列表去选择单个元素，以下两种方式效果一样：

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

图4-1可以展示二维数组上的索引，我们可以将0轴看作“行”，将1轴看作“列”：

		axis 1		
		0	1	2
	0	0, 0	0, 1	0, 2
axis 0	1	1, 0	1, 1	1, 2
	2	2, 0	2, 1	2, 2

图4-1：在一个Numpy数组中对元素进行索引

在多维数组中，你可以省略后续索引值，返回的对象将是降低一个维度的数组。因此在一个2×2×3的数组arr3d中：

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 1

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

arr3d[0]是一个2×3的数组：

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

标量和数组都可以传递给arr3d[0]：

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
```

```
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

类似地，arr3d[1，0]返回的是一个一维数组：

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

上面的表达式可以分解为下面两步：

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

需要注意的是，以上的数组子集选择中，返回的数组都是视图

4.1.4.1 数组的切片索引

与Python列表的一维对象类似，数组可以通过类似的语法进行切片：

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [89]: arr[1:6]
Out[89]: array([ 1,  2,  3,  4, 64])
```

再回想下前面的二维数组，arr2d，对数组进行切片略有不同：

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])
```

如你所见，数组沿着轴0进行了切片。表达式arrzd[：2]的含义为选择arr2d的前两“行”。

你可以进行多组切片，与多组索引类似：

```
In [92]: arr2d[:2, 1:]
Out[92]:
array([[2, 3],
       [5, 6]])
```

当你像在上面这个例子中那样切片时，你需要按照原数组的维度进行切片。如果将索引和切片混合，就可以得到低维度的切片。

例如，我可以选择第二行但是只选择前两列：

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

类似地，我也可以选择第三列，但是只选择前两行：

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

如图4-2所示。需要注意的是，单独一个冒号表示选择整个轴上的数组，因此你可以按照下面的方式在更高维度上进行切片：

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
```

```
        [7]])
```

当然对切片表达式赋值时，整个切片都会重新赋值：

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)
arr[:, :2]	(3, 2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

图4-2：二维数组的切片

4.1.5　布尔索引

让我们考虑以下例子，假设我们的数据都在数组中，并且数组中的数据是一些存在重复的人名。我会使用numpy.random中的randn函数来生成一些随机正态分布的数据：

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

假设每个人名都和data数组中的一行相对应，并且我们想要选中所有'Bob'对应的行。与数学操作类似，数组的比较操作（比如＝＝）也是可以向量化的。因此，比较names数组和字符串'Bob'会产生一个布尔值数组：

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], d
```

在索引数组时可以传入布尔值数组：

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

布尔值数组的长度必须和数组轴索引长度一致。你甚至还可以用切片

或整数值（或整数值的序列，后续会介绍）对布尔值数组进行混合和匹配。

当布尔值数组的长度不正确时，布尔值选择数据的方法并不会报错，因此我建议在使用该特性的时候要小心。

在这些例子中，我选择了names = = 'Bob'的行，并索引了各个列：

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

为了选择除了'Bob'以外的其他数据，你可以使用！=或在条件表达式前使用~对条件取反：

```
In [106]: names != 'Bob'
Out[106]: array([False,  True,  True, False,  True,  True,  True], dt

In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

~符号可以在你想要对一个通用条件进行取反时使用：

```
In [108]: cond = names == 'Bob'
In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

当要选择三个名字中的两个时，可以对多个布尔值条件进行联合，需要使用数学操作符如&（and）和|（or）：

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dt

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

使用布尔值索引选择数据时，总是生成数据的拷贝，即使返回的数组并没有任何变化。

 Python的关键字and和or对布尔值数组并没有用，请使用&（and）和|（or）来代替。

基于常识来设置布尔值数组的值也是可行的。将data中所有的负值设置为0，我们需要做：

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.    ,  0.    ],
       [ 1.669 ,  0.    ,  0.    ,  0.477 ],
       [ 3.2489,  0.    ,  0.    ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

利用一维布尔值数组对每一行设置数值也是非常简单的：

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
```

```
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

后续章节我们将会看到，如何利用pandas方便地在二维数据上进行上述的操作。

4.1.6 神奇索引

神奇索引是NumPy中的术语，用于描述使用整数数组进行数据索引。

假设我们有一个8×4的数组：

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

为了选出一个符合特定顺序的子集，你可以简单地通过传递一个包含指明所需顺序的列表或数组来完成：

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

代码运行出你所想要的结果！如果使用负的索引，将从尾部进行选择：

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

传递多个索引数组时情况有些许不同，这样会根据每个索引元组对应

的元素选出一个一维数组：

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

在附录A中可以看到reshape方法的更多细节。

在上述例子中，元素（1，0）、（5，3）、（7，1）和（2，2）被选中。如果不考虑数组的维数（本例中是二维），神奇索引的结果总是一维的。

在本例中，神奇索引的行为和一些用户所设想的并不相同（包括我自己）。通常情况下，我们所设想的结果是通过选择矩阵中行列的子集所形成的矩形区域。下面是实现我们想法的一种方式：

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

请牢记神奇索引与切片不同，它总是将数据复制到一个新的数组中。

4.1.7 数组转置和换轴

转置是一种特殊的数据重组形式，可以返回底层数据的视图而不需要复制任何内容。数组拥有transpose方法，也有特殊的T属性：

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

当进行矩阵计算时，你可能会经常进行一些特定操作，比如，当计算矩阵内积会使用np.dot：

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

对于更高维度的数组，transpose方法可以接收包含轴编号的元组，用于置换轴（拓展下思维）：

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

在这里，轴已经被重新排序，使得原先的第二个轴变为第一个，原先的第一个轴变成了第二个，最后一个轴并没有改变。

使用.T进行转置是换轴的一个特殊案例。ndarray有一个swapaxes方法，该方法接收一对轴编号作为参数，并对轴进行调整用于重组数据：

```
In [135]: arr
Out[135]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

swapaxes返回的是数据的视图，而没有对数据进行复制。

4.2　通用函数：快速的逐元素数组函数

通用函数，也可以称为ufunc，是一种在ndarray数据中进行逐元素操作的函数。某些简单函数接收一个或多个标量数值，并产生一个或多个标量结果，而通用函数就是对这些简单函数的向量化封装。

有很多ufunc是简单的逐元素转换，比如sqrt或exp函数：

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

这些是所谓的一元通用函数。还有一些通用函数，比如add或maximum则会接收两个数组并返回一个数组作为结果，因此称为二元通用函数：

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

这里，numpy.maximum逐个元素地将x和y中元素的最大值计算出来。

也有一些通用函数返回多个数组。比如modf，是Python内建函数divmod的向量化版本。它返回了一个浮点值数组的小数部分和整数部分：

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

通用函数接收一个可选参数out，允许对数组按位置操作：

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 , 5.3731, 3.6182, 3.45 , 5

In [152]: np.sqrt(arr)
Out[152]: array([ nan,  nan,  nan,  2.318,  1.9022,  1.8574,  2.2378

In [153]: np.sqrt(arr, arr)
Out[153]: array([ nan,  nan,  nan,  2.318,  1.9022,  1.8574,  2.2378

In [154]: arr
Out[154]: array([ nan,  nan,  nan,  2.318,  1.9022,  1.8574,  2.2378
```

表4-3和表4-4列举的是可用的通用函数

表4-3：一元通用函数

函数名	描述
abs、fabs	逐元素地计算整数、浮点数或复数的绝对值
sqrt	计算每个元素的平方根（与 arr ** 0.5 相等）
square	计算每个元素的平方（与 arr**2 相等）

表4-3：一元通用函数（续）

函数名	描述
exp	计算每个元素的自然指数值 e^x
log、log10、log2、log1p	分别对应：自然对数（e 为底）、对数 10 为底、对数 2 为底、log（1+x）
sign	计算每个元素的符号值：1（正数）、0（0）、-1（负数）
ceil	计算每个元素的最高整数值（即大于等于给定数值的最小整数）
floor	计算每个元素的最小整数值（即小于等于给定元素的最大整数）
rint	将元素保留到整数位，并保持 dtype
modf	分别将数组的小数部分和整数部分按数组形式返回
isnan	返回数组中的元素是否是一个 NaN（不是一个数值），形式为布尔值数组
isfinite、isinf	分别返回数组中的元素是否有限（非 inf、非 NaN）、是否无限的，形式为布尔值数组
cos、cosh、sin、sinh、tan、tanh	常规的双曲三角函数
arccos、arccosh、arcsin、arcsinh、arctan、arctanh	反三角函数
logical_not	对数组的元素按位取反（与 ~arr 效果一致）

表4-4：二元通用函数

函数名	描述
add	将数组的对应元素相加
subtract	在第二个数组中，将第一个数组中包含的元素去除
multiply	将数组的对应元素相乘
divide, floor_divide	除或整除（放弃余数）
power	将第二个数组的元素作为第一个数组对应元素的幂次方
maximum, fmax	逐个元素计算最大值，fmax 忽略 NaN
minimum, fmin	逐个元素计算最小值，fmin 忽略 NaN
mod	按元素的求模计算（即求除法的余数）
copysign	将第一个数组的符号值改为第二个数组的符号值
greater,greater_equal,less,less_equal,equal,not_equal	进行逐个元素的比较，返回布尔值数组（与数学操作符 >、>=、<、<=、==、!= 效果一致）
logical_and,logical_or,logical_xor	进行逐个元素的逻辑操作（与逻辑操作符 &、|、^ 效果一致）

4.3 使用数组进行面向数组编程

使用NumPy数组可以使你利用简单的数组表达式完成多种数据操作任务，而无须写些大量循环。这种利用数组表达式来替代显式循环的方法，称为向量化。通常，向量化的数组操作会比纯Python的等价实现在速度上快一到两个数量级（甚至更多），这对所有种类的数值计算产生了最大的影响。附录A中我解释的广播机制，就是向量化计算的有效方式。

作为一个简单的示例，假设我们想要对一些网格数据来计算函数sqrt（x^2＋y^2）的值。np.meshgrid函数接收两个一维数组，并根据两个数组的所有（x，y）对生成一个二维矩阵：

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced point

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

现在，你可以用和两个坐标值同样的表达式来使用函数：

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

作为第9章的前瞻，我使用matplotlib来生成这个二维数组的可视化：

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar at 0x7f715e3fa630>

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of v
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

在图4-3中，我使用了matplotlib函数imshow根据函数值的二维数组生成一个图像。

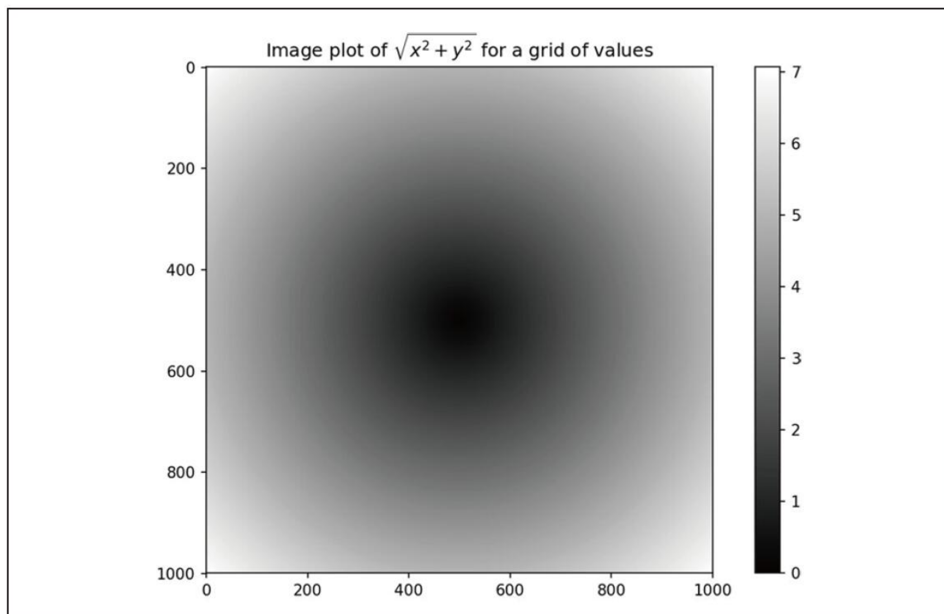


图4-3：函数在网格数据上的结果图像

4.3.1 将条件逻辑作为数组操作

numpy.where函数是三元表达式x if condition else y的向量化版本。假设我们有一个布尔值数组和两个数值数组：

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

假设cond中的元素为True时，我们取xarr中的对应元素值，否则取yarr中的元素。我们可以通过列表推导式来完成，像下列代码这样：

```
In [168]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999
```

这样会产生多个问题。首先，如果数组很大的话，速度会很慢（因为所有的工作都是通过解释器解释Python代码完成）。其次，当数组是多维时，就无法凑效了。而使用np.where时，就可以非常简单地完成：

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

np.where的第二个和第三个参数并不需要是数组，它们可以是标量。where在数据分析中的一个典型用法是根据一个数组来生成一个新的数组。假设你有一个随机生成的矩阵数据，并且你想将其中的正值都替换为2，将所有的负值替换为-2，使用np.where会很容易实现：

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
```

```
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

你可以使用np.where将标量和数组联合，例如，我可以像下面的代码那样将arr中的所有正值替换为常数2：

```
In [176]: np.where(arr > 0, 2, arr) # 仅将正值设为2
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

传递给np.where的数组既可以是同等大小的数组，也可以是标量。

4.3.2 数学和统计方法

许多关于计算整个数组统计值或关于轴向数据的数学函数，可以作为数组类型的方法被调用。你可以使用聚合函数（通常也叫缩减函数），比如sum、mean和std（标准差），既可以直接调用数组实例的方法，也可以使用顶层的NumPy函数。

此处我生成了一些正态分布的随机数，并计算了部分聚合统计数据：

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

像mean、sum等函数可以接收一个可选参数axis，这个参数可以用于计算给定轴向上的统计值，形成一个下降一维度的数组：

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

arr.mean（1）表示“计算每一列的平均值”，而arr.sum（0）表示“计算行轴向的累和”。

其他的方法，例如cumsum和cumprod并不会聚合，它们会产生一个中间结果：

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

在多维数组中，像cumsum这样的累积函数返回相同长度的数组，但是可以在指定轴向上根据较低维度的切片进行部分聚合：

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

表4-5是统计方法的列表，我们将会在后面的章节中看到这些方法的大量示例。

表4-5：基础数组统计方法

方法	描述
sum	沿着轴向计算所有元素的累和，0 长度的数组，累和为 0
mean	数学平均，0 长度的数组平均值为 NaN
std, var	标准差和方差，可以选择自由度调整（默认分母是 n）
min, max	最小值和最大值
argmin, argmax	最小值和最大值的位置
cumsum	从 0 开始元素累积和
cumprod	从 1 开始元素累积积

4.3.3 布尔值数组的方法

在前面介绍的方法，布尔值会被强制为1（True）和0（False）。因此，sum通常可以用于计算布尔值数组中的True的个数：

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # 正值的个数
Out[191]: 42
```

对于布尔值数组，有两个非常有用的方法any和all。any检查数组中是否至少有一个True，而all检查是否每个值都是True：

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

这些方法也可适用于非布尔值数组，所有的非0元素都会按True处理。

4.3.4 排序

和Python的内建列表类型相似，NumPy数组可以使用sort方法按位置排序：

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938,  1.24  , -0.1357,  1.43  , -0.846

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,  1.43
```

你可以在多维数组中根据传递的axis值，沿着轴向对每个一维数据段进行排序：

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

顶层的np.sort方法返回的是已经排序好的数组拷贝，而不是对原数组按位置排序。下面的例子计算的是一个数组的分位数，并选出分位数所对应的值，这是一种应急的方式：

```
In [203]: large_arr = np.random.randn(1000)
```

```
In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[205]: -1.5311513550102103
```

更多使用NumPy排序方法的细节，以及像间接排序这类更高级的技术，请参考附录A。一些其与排序相关的其他种类的数据操作（比如根据一张表中的某一列进行排序），也将会出现在pandas中。

4.3.5 唯一值与其他集合逻辑

NumPy包含一些针对一维ndarray的基础集合操作。常用的一个方法是np.unique，返回的是数组中唯一值排序后形成的数组：

```
In [206]: names =np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

将np.unique和纯Python实现相比较：

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

另一个函数，np.in1d，可以检查一个数组中的值是否在另外一个数组中，并返回一个布尔值数组：

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dt
```

表4-6是NumPy中集合函数的列表。

表4-6：数组的集合操作

方法	描述
`unique(x)`	计算 x 的唯一值，并排序
`intersect1d(x, y)`	计算 x 和 y 的交集，并排序
`union1d(x, y)`	计算 x 和 y 的并集，并排序
`in1d(x, y)`	计算 x 中的元素是否包含在 y 中，返回一个布尔值数组
`setdiff1d(x, y)`	差集，在 x 中但不在 y 中的 x 的元素
`setxor1d(x, y)`	异或集，在 x 或 y 中，但不属于 x、y 交集的元素

4.4 使用数组进行文件输入和输出

NumPy可以在硬盘中将数据以文本或二进制文件的形式进行存入硬盘或由硬盘载入。在本节，我将只讨论NumPy的内建二进制格式，因为大部分用户更倾向于使用pandas或其他工具来载入文本或表格型数据（在第6章将会看到更多）。

np.save和np.load是高效存取硬盘数据的两大工具函数。数组在默认情况下是以未压缩的格式进行存储的，后缀名是.npy：

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

如果文件存放路径中没写.npy时，后缀名会被自动加上。硬盘上的数组可以使用np.load进行载入：

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

你可以使用np.savez并将数组作为参数传递给该函数，用于在未压缩文件中保存多个数组：

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

当载入一个.npy文件的时候，你会获得一个字典型的对象，并通过该对象很方便地载入单个数组：

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

如果你的数据已经压缩好了，你可能会想要使用numpy.savez_compressed将数据存入已经压缩的文件：

numpy.savez_compressed将数据存入已经压缩的文件：

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

4.5 线性代数

线性代数，比如矩阵乘法、分解、行列式等方阵数学，是所有数组类库的重要组成部分。和Matlab等其他语言相比，NumPy的线性代数中所不同的是*是矩阵的逐元素乘积，而不是矩阵的点乘积。因此NumPy的数组方法和numpy命名空间中都有一个函数dot，用于矩阵的操作：

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [227]: x.dot(y)
Out[227]:
array([[  28.,   64.],
       [  67.,  181.]])
```

x.dot（y）等价于np.dot（x，y）：

```
In [228]: np.dot(x, y)
Out[228]:
array([[  28.,   64.],
       [  67.,  181.]])
```

一个二维数组和一个长度合适的一维数组之间的矩阵乘积，其结果是一个一维数组：

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

特殊符号@也作为中缀操作符，用于点乘矩阵操作：

```
In [230]: x @ np.ones(3)
Out[230]: array([  6.,  15.])
```

numpy.linalg拥有一个矩阵分解的标准函数集，以及其他常用函数，例如求逆和行列式求解。这些函数都是通过在MATLAB和R等其他语言使用的相同的行业标准线性代数库来实现的，例如BLAS、LAPACK或英特尔专有的MKL（数学核心库）（是否使用MKL取决于使用NumPy的版本）：

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[  933.1189,   871.8258, -1417.6902, -1460.4005,  1782.1391],
       [  871.8258,   815.3929, -1325.9965, -1365.9242,  1666.9347],
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6822],
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.422 ],
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.5128]])

In [235]: mat.dot(inv(mat))
Out[235]:
array([[ 1.,  0., -0., -0., -0.],
       [-0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [-0.,  0.,  0.,  1., -0.],
       [-0.,  0.,  0.,  0.,  1.]])

In [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914,  4.38  ,  0.1757,  0.4075, -0.7838],
       [ 0.    , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0.    ,  0.    , -0.8138,  1.5414,  0.6155],
       [ 0.    ,  0.    ,  0.    , -2.6445, -2.1669],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.0002]])
```

表达式X.T.dot（X）计算的是X和它的转置矩阵X.T的点乘积。

表4-7是最常用的线性代数函数列表。

表4-7：常用numpy.linalg函数

函数	描述
diag	将一个方阵的对角（或非对角）元素作为一维数组返回，或者将一维数组转换成一个方阵，并且在非对角线上有零点
dot	矩阵点乘
trace	计算对角元素和
det	计算矩阵的行列式
eig	计算方阵的特征值和特征向量
inv	计算方阵的逆矩阵
pinv	计算矩阵的 Moore-Penrose 伪逆
qr	计算 QR 分解
svd	计算奇异值分解（SVD）
solve	求解 x 的线性系统 Ax = b，其中 A 是方阵
lstsq	计算 Ax = b 的最小二乘解

4.6 伪随机数生成

numpy.random模块填补了Python内建的random模块的不足，可以高效地生成多种概率分布下的完整样本值数组。例如，你可以使用normal来获得一个4×4的正态分布样本数组：

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

然而Python内建的random模块一次只能生成一个值。你可以从下面的示例中看到，numpy.random在生成大型样本时比纯Python的方式快了一个数量级：

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops ea
```

我们可以称这些为伪随机数，因为它们是由具有确定性行为的算法根据随机数生成器中的随机数种子生成的。你可以通过np.random.seed更改NumPy的随机数种子：

```
In [244]: np.random.seed(1234)
```

numpy.random中的数据生成函数公用了一个全局的随机数种子。为了避免全局状态，你可以使用numpy.random.RandomState生成一个随机数生成器，使数据独立于其他的随机数状态：

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

表4-8中是numpy.random中可用的部分函数。我将会在下一节中借助这些函数来一次性生成大型样本数组

表4-8：numpy.random中的部分函数列表

函数	描述
seed	向随机数生成器传递随机状态种子
permutation	返回一个序列的随机排列，或者返回一个乱序的整数范围序列
shuffle	随机排列一个序列
rand	从均匀分布中抽取样本
randint	根据给定的由低到高的范围抽取随机整数
randn	从均值 0 方差 1 的正态分布中抽取样本（MATLAB 型接口）
binomial	从二项分布中抽取样本
normal	从正态（高斯）分布中抽取样本
beta	从 beta 分布中抽取样本
chisquare	从卡方分布中抽取样本

表4-8：numpy.random中的部分函数列表（续）

函数	描述
gamma	从伽马分布中抽取样本
uniform	从均匀 [0,1) 分布中抽取样本

4.7 示例：随机漫步

随机漫步的模拟（https://en.wikipedia.org/wiki/Random_walk ）提供了一个使用数组操作的说明性应用。首先，让我们考虑一个简单的随机漫步，从0开始，步进为1和-1，且两种步进发生的概率相等。

以下是使用内建random模块利用纯Python实现的一个1，000步的随机漫步：

```
1,000步的随机漫步：
In [247]: import random
   .....: position = 0
   .....: walk = [position]
   .....: steps = 1000
   .....: for i in range(steps):
   .....:     step = 1 if random.randint(0, 1) else -1
   .....:     position += step
   .....:     walk.append(position)
   .....:
```

图4-4是对上面随机漫步的前100步的可视化：

```
In [249]: plt.plot(walk[:100])
```

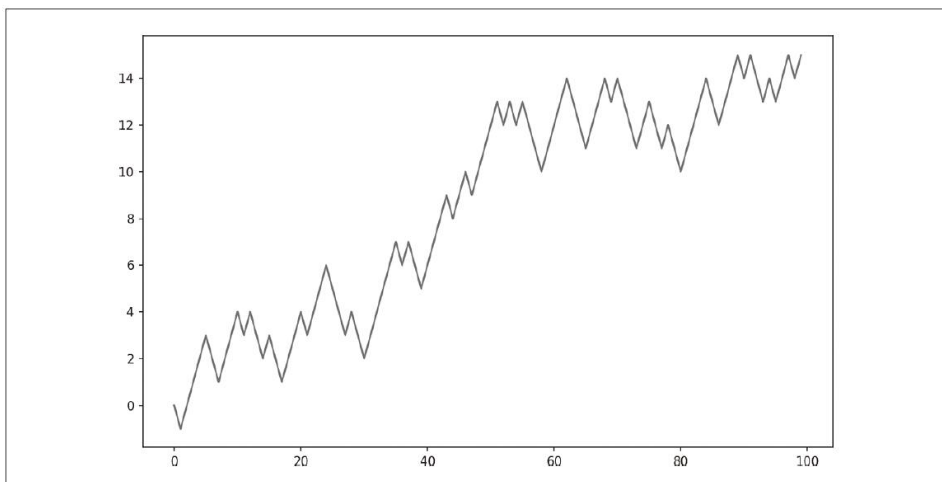


图4-4：简单的随机漫步

你可能会观察到walk只是对随机步进的累积，并且可以通过一个数组表达式实现。因此，我使用np.random模块一次性抽取1，000次投掷硬币的结果，每次投掷的结果为1或-1，然后计算累积值：

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()
```

由此我们开始从漫步轨道上提取一些统计数据，比如最大值、最小值等：

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

更复杂的统计是第一次穿越时间，即随机漫步的某一步达到了某个特定值。这里假设我们想要知道漫步中是何时连续朝某个方向连续走了10步。np.abs（walk）>＝10给我们一个布尔值数组，用于表明漫步是否连续在同一方向走了十步，但是我们想要的是第一次走了10步或-10步的位置。我们可以使用argmax来计算，该函数可以返回布尔值数组中最大值的第一个位置（True就是最大值）：

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

请注意，这里使用argmax效率并不高，因为它总是完整地扫描整个数组。在这个特殊的示例中，一旦True被发现，我们就知道最大值了。

4.7.1 一次性模拟多次随机漫步

如果你的目标是模拟多次随机漫步，比如说5，000步。你可以稍微地修改下之前的代码来生成所有的随机步。如果传入一个2个元素的元组，numpy.random中的函数可以生成一个二维的抽取数组，并且我们可以一次性地跨行算出全部5，000个随机步的累积和：

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)
In [263]: walks
Out[263]:
array([[  1,   0,   1, ...,   8,   7,   8],
       [  1,   0,  -1, ...,  34,  33,  32],
       [  1,   0,  -1, ...,   4,   5,   4],
       ...,
       [  1,   2,   1, ...,  24,  25,  26],
       [  1,   2,   3, ...,  14,  13,  14],
       [ -1,  -2,  -3, ..., -24, -23, -22]])
```

现在我们可以计算出这些随机步的最大值和最小值了：

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

让我们在这些随机步中计算出30或-30的最小穿越时间。这有点棘手，因为不是所有的5，000个都达到了30。我们可以用any方法来检查：

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False, True, False, ..., False, True, False], dtype=

In [268]: hits30.sum() # 达到30或-30的数字
```

```
Out[268]: 3410
```

我们可以使用布尔值数组来选出绝对步数超过30的步所在的行，并使用argmax从轴向1上获取穿越时间：

```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

利用其他分布而不是等概率的掷硬币实验来随机漫步也是很容易的。你只需要使用一个不同的随机数生成函数，比如normal，再根据特定的均值和标准差即可生成正态分布下的随机步：

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```

4.8 本章小结

本书的后续章节的大部分，都会集中在打造pandas数据处理技能上，我们将会继续在基于数组的风格下处理数据。在附录A中，我们将深挖NumPy的特性，帮助你进一步提高数组计算技能。

第5章　pandas入门

pandas是贯穿本书后续部分的主要工具。它所包含的数据结构和数据处理工具的设计使得在Python中进行数据清洗和分析非常快捷。pandas经常是和其他数值计算工具，比如NumPy和SciPy，以及数据可视化工具比如matplotlib一起使用的。pandas支持大部分NumPy语言风格的数组计算，尤其是数组函数以及没有for循环的各种数据处理。

尽管pandas采用了很多NumPy的代码风格，但最大的不同在于pandas是用来处理表格型或异质型数据的。而NumPy则相反，它更适合处理同质型的数值类数组数据。

由于在2010年成为了开源项目，pandas已经逐渐成熟，成为一个在真实世界中广泛应用的大型类库。pandas的开发者社区已经有超过800个代码贡献者，他们帮助构建了项目，并将pandas应用到日常中去解决他们的数据难题。

贯穿本书的后续章节，我会使用下面的便捷方式导入pandas：

```
In [1]: import pandas as pd
```

因此，无论何时，只要你在代码中看到pd.，它表示对pandas的引用。你还可以方便地从本地命名空间中导入Series和DataFrame，它们是常用的类：

```
In [2]: from pandas import Series, DataFrame
```

5.1 pandas数据结构介绍

为了入门pandas，你需要熟悉两个常用的工具数据结构：Series和DataFrame。尽管他们并不能解决所有的问题，但它们为大多数应用提供了一个有效、易用的基础。

5.1.1 Series

Series是一种一维的数组型对象，它包含了一个值序列（与NumPy中的类型相似），并且包含了数据标签，称为索引（index）。最简单的序列可以仅仅由一个数组形成：

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

交互式环境中Series的字符串表示，索引在左边，值在右边。由于我们不为数据指定索引，默认生成的索引是从0到N-1（N是数据的长度）。你可以通过values属性和index属性分别获得Series对象的值和索引：

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # 与 range(4)类似
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

通常需要创建一个索引序列，用标签标识每个数据点：

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

与NumPy的数组相比，你可以在从数据中选择数据的时候使用标签来

进行索引：

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

上面的例子中，['c'，'a'，'d']包含的不是数字而是字符串，作为索引列表。

使用NumPy的函数或NumPy风格的操作，比如使用布尔值数组进行过滤，与标量相乘，或是应用数学函数，这些操作将保存索引值连接：

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

从另一个角度考虑Series，可以认为它是一个长度固定且有序的字典，因为它将索引值和数据值按位置配对。在你可能会使用字典的上下文中，也可以使用Series：

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

如果你已经有数据包含在Python字典中，你可以使用字典生成一个Series：

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Ut

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

当你把字典传递给Series构造函数时，产生的Series的索引将是排序好的字典键。你可以将字典键按照你所想要的顺序传递给构造函数，从而使生成的Series的索引顺序符合你的预期：

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']
In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

上面的例子中，sdata中的三个值被放置在正确的位置，但是因为'California'没有出现在sdata的键中，它对应的值是NaN（not a number），这是pandas中标记缺失值或NA值的方式。因为'Utah'并不在states中，它被排除在结果对象外。

我将持续使用术语“缺失”或“NA”来表示缺失数据。pandas中使用isnull和notnull函数来检查缺失数据：

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

isnull和notnull也是Series的实例方法：

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

第7章中我们会更深入地讨论处理缺失数据的细节。

对于很多应用来说，在数学操作中自动对齐索引是Series的一个非常有用的特性：

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California          NaN
Ohio            70000.0
Oregon          32000.0
Texas          142000.0
Utah                NaN
dtype: float64
```

数据对齐特性将在后续章节深入讨论。如果你有数据库经验，你可以认为这个特性与数据库的join操作是非常相似的。

Series对象自身和其索引都有name属性，这个特性与pandas其他重要功能集成在一起：

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California         NaN
Ohio           35000.0
Oregon         16000.0
Texas          71000.0
Name: population, dtype: float64
```

Series的索引可以通过按位置赋值的方式进行改变：

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

5.1.2 DataFrame

DataFrame表示的是矩阵的数据表，它包含已排序的列集合，每一列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame既有行索引也有列索引，它可以被视为一个共享相同索引的Series的字典。在DataFrame中，数据被存储为一个以上的二维块，而不是列表、字典或其他一维数组的集合。DataFrame内部的精确细节不在本书的讨论范围。

 尽管DataFrame是二维的，但你可以利用分层索引在DataFrame中展现更高维度的数据。分层索引是pandas中一种更为高级的数据处理特性，我们将在第8章中进行讨论。

有多种方式可以构建DataFrame，其中最常用的方式是利用包含等长度列表或NumPy数组的字典来形成DataFrame：

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

产生的DataFrame会自动为Sereies分配索引，并且列会按照排序的顺序排列：

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

如果你使用的是Jupyter notebook，pandas的DataFrame对象将会展示为一个对浏览器更为友好的HTML表格。

对于大型DataFrame，head方法将会只选出头部的五行：

```
In [46]: frame.head()
```

```
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

如果你指定了列的顺序，DataFrame的列将会按照指定顺序排列：

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

如果你传的列不包含在字典中，将会在结果中出现缺失值：

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop'
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])

In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame中的一列，可以按字典型标记或属性那样检索为Series：

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
```

```
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

在IPython中，属性型连接（比如frame2.year）和列名的tab补全是非常方便的。

frame2[colunm]对于任意列名均有效，但是frame2.column只在列名是有效的Python变量名时有效。

请注意，返回的Series与原DataFrame有相同的索引，且Series的name属性也会被合理地设置。

行也可以通过位置或特殊属性loc进行选取（后续会详细介绍）：

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

列的引用是可以修改的。例如，空的'debt'列可以赋值为标量值或值数组：

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
```

```
five    2002  Nevada  2.9  16.5
six     2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

当你将列表或数组赋值给一个列时，值的长度必须和DataFrame的长度相匹配。如果你将Series赋值给一列时，Series的索引将会按照DataFrame的索引重新排列，并在空缺的地方填充缺失值：

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', '

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

如果被赋值的列并不存在，则会生成一个新的列。del关键字可以像在字典中那样对DataFrame删除列。

在del的例子中，我首先增加一列，这一列是布尔值，判断条件是state列是否为'ohio'：

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
```

```
four    2001  Nevada  2.4  -1.5    False
five    2002  Nevada  2.9  -1.7    False
six     2003  Nevada  3.2   NaN    False
```

 frame2.eastern的语法无法创建新的列。

del方法可以用于移除之前新建的列：

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

 从DataFrame中选取的列是数据的视图，而不是拷贝。因此，对Series的修改会映射到DataFrame中。如果需要复制，则应当显式地使用Series的copy方法。

另一种常用的数据形式是包含字典的嵌套字典

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

如果嵌套字典被赋值给DataFrame，pandas会将字典的键作为列，将内部字典的键作为行索引：

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

你可以将使用类似NumPy的语法对DataFrame进行转置操作（调换行和列）：

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
```

```
Nevada    NaN    2.4    2.9
Ohio      1.5    1.7    3.6
```

内部字典的键被联合、排序后形成了结果的索引。如果已经显式指明索引的话，内部字典的键将不会被排序：

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

包含Series的字典也可以用于构造DataFrame：

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

可以向DataFrame构造函数传递的对象列表见表5-1。

如果DataFrame的索引和列拥有name属性，则这些name属性也会被显示：

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'
In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

和Series类似，DataFrame的values属性会将包含在DataFrame中的数据以二维ndarray的形式返回：

```
In [74]: frame3.values
```

```
Out[74]:
array([[ nan,   1.5],
       [ 2.4,   1.7],
       [ 2.9,   3.6]])
```

如果DataFrame的列是不同的dtypes，则values的dtype会自动选择适合所有列的类型：

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

表5-1：DataFrame构造函数的有效输入

类型	注释
2D ndarray	数据的矩阵，行和列的标签是可选参数
数组、列表和元组构成的字典	每个序列成为 DataFrame 的一列，所有的序列必须长度相等
NumPy 结构化 / 记录化数组	与数组构成的字典一致
Series 构成的字典	每个值成为一列，每个 Series 的索引联合起来形成结果的行索引，也可以显式地传递索引
字典构成的字典	每一个内部字典成为一列，键联合起来形成结果的行索引
字典或 Series 构成的列表	列表中的一个元素形成 DataFrame 的一行，字典键或 Series 索引联合起来形成 DataFrame 的列标签
列表或元组构成的列表	与 2D ndarray 的情况一致
其他 DataFrame	如果不显示传递索引，则会使用原 DataFrame 的索引
NumPy MaskedArray	与 2D ndarray 的情况类似，但隐蔽值会在结果 DataFrame 中成为 NA/ 缺失值

5.1.3 索引对象

pandas中的索引对象是用于存储轴标签和其他元数据的（例如轴名称或标签）。在构造Series或DataFrame时，你所使用的任意数组或标签序列都可以在内部转换为索引对象：

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

索引对象是不可变的，因此用户是无法修改索引对象的：

```
index[1] = 'd'  # TypeError
```

不变性使得在多种数据结构中分享索引对象更为安全：

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

一些用户并不经常利用索引对象提供的功能，但是因为一些操作会产生包含索引化数据的结果，理解索引如何工作还是很重要的。

除了类似数组，索引对象也像一个固定大小的集合：

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

与Python集合不同，pandas索引对象可以包含重复标签：

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

根据重复标签进行筛选，会选取所有重复标签对应的数据。

每个索引都有一些集合逻辑的方法和属性，这些方法和属性解决了关于它所包含的数据的其他常见问题。表5-2中总结了这些方法和属性中常用的一部分。

表5-2：一些索引对象的方法和属性

方法	描述
append	将额外的索引对象粘贴到原索引后，产生一个新的索引
difference	计算两个索引的差集
intersection	计算两个索引的交集
union	计算两个索引的并集
isin	计算表示每一个值是否在传值容器中的布尔数组
delete	将位置 i 的元素删除，并产生新的索引
drop	根据传参删除指定索引值，并产生新的索引
insert	在位置 i 插入元素，并产生新的索引
is_monotonic	如果索引序列递增则返回 True
is_unique	如果索引序列唯一则返回 True
unique	计算索引的唯一值序列

5.2 基本功能

本节将会指引你了解与Series或DataFrame中数据交互的基础机制。后续的章节中会更为深入地讲解使用pandas进行数据分析和操作的主题。本书并不是pandas的详尽文档，我们关注的是最重要的特性，而将不那么常用（或者说更为深奥）的特性留给读者自行探索。

5.2.1 重建索引

reindex是pandas对象的重要方法，该方法用于创建一个符合新索引的新对象。考虑下面的例子：

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a'

In [92]: obj
Out[92]:
d 4.5
b 7.2
a -5.3
c 3.6
dtype: float64
```

Series调用reindex方法时，会将数据按照新的索引进行排列，如果某个索引值之前并不存在，则会引入缺失值：

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a    -5.3
b     7.2
c     3.6
d     4.5
e     NaN
dtype: float64
```

对于顺序数据，比如时间序列，在重建索引时可能会需要进行插值或填值。method可选参数允许我们使用诸如ffill等方法在重建索引时插值，ffill方法会将值前向填充：

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2,
In [96]: obj3
Out[96]:
0      blue
2    purple
4    yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
```

```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

在DataFrame中，reindex可以改变行索引、列索引，也可以同时改变二者。当仅传入一个序列时，结果中的行会重建索引：

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2
Out[101]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

列可以使用columns关键字重建索引：

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

表5-3是reindex方法的参数列表。

我们更深入地探索时，你可以使用loc进行更为简洁的标签索引，许多

用户更倾向于使用这种方式：

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
   Texas  Utah  California
a    1.0   NaN         2.0
b    NaN   NaN         NaN
c    4.0   NaN         5.0
d    7.0   NaN         8.0
```

表5-3：reindex方法的参数

参数	描述
index	新建作为索引的序列，可以是索引实例或任意其他序列型 Python 数据结构，索引使用时无须复制

表5-3：reindex方法的参数（续）

参数	描述
method	插值方式；'ffill' 为前向填充，而 'bfill' 是后向填充
fill_value	通过重新索引引入缺失数据时使用的替代值
limit	当前向或后向填充时，所需填充的最大尺寸间隙（以元素数量）
tolerance	当前向或后向填充时，所需填充的不精确匹配下的最大尺寸间隙（以绝对数字距离）
level	匹配 MultiIndex 级别的简单索引；否则选择子集
copy	如果为 True，即使新索引等于旧索引，也总是复制底层数据；如果是 False，则在索引相同时不要复制数据

5.2.2　轴向上删除条目

如果你已经拥有索引数组或不含条目的列表，在轴向上删除一个或更多的条目就非常容易，但这样需要一些数据操作和集合逻辑，drop方法会返回一个含有指示值或轴向上删除值的新对象：

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd',

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

在DataFrame中，索引值可以从轴向上删除。为了表明这个特性，我们首先创建一个示例DataFrame：

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
Utah          8    9      10     11
New York     12   13      14     15
```

在调用drop时使用标签序列会根据行标签删除值（轴0）：

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

你可以通过传递axis=1或axis='columns'来从列中删除值：

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

很多函数，例如drop，会修改Series或DataFrame的尺寸或形状，这些方法直接操作原对象而不返回新对象：

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

请注意inplace属性，它会清除被删除的数据。

5.2.3 索引、选择与过滤

Series的索引（obj[...]）与NumPy数组索引的功能类似，只不过Series的索引值可以不仅仅是整数。相关示例如下：

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

普通的Python切片中是不包含尾部的，Series的切片与之不同：

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

使用这些方法设值时会修改Series相应的部分：

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

使用单个值或序列，可以从DataFrame中索引出一个或多个列：

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [130]: data['two']
Out[130]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

这种索引方式也有特殊案例。首先，可以根据一个布尔值数组切片或选择数据：

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [133]: data[data['three'] > 5]
Out[133]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

行选择语法data[：2]非常方便。传递单个元素或一个列表到[]符号中可以选择列。

另一个用例是使用布尔值DataFrame进行索引，布尔值DataFrame可以是对标量值进行比较产生的：

```
In [134]: data < 5
Out[134]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

在这个特殊例子中，这种索引方式使得DataFrame在语法上更像是NumPy二维数组。

5.2.3.1 使用loc和iloc选择数据

针对DataFrame在行上的标签索引，我将介绍特殊的索引符号loc和

iloc。他们允许你使用轴标签（loc）或整数标签（iloc）以NumPy风格的语法从DataFrame中选出数组的行和列的子集。

我们通过标签选出单行多列的数据作为基础示例：

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

然后我们使用整数标签iloc进行类似的数据选择：

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

除了单个标签或标签列表之外，索引功能还可以用于切片：

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
```

```
Colorado     0     5      6
Utah         8     9     10
New York    12    13     14
```

因此，有多种方式可以选择、重排pandas对象中的数据。表5-4提供了DataFrame选择数据的一些方法。之后的章节会介绍处理分层索引的可选参数。

在早期设计pandas时，我认为使用frame[:，col]来选择一列太过冗余（也容易出错），这是因为列选择是最为常用的操作。我做了设计权衡，将所有的神奇索引行为（包括标签和整数）都放到ix操作符中。事实上，这导致了整数轴标签数据中的许多边缘情况，所以pandas团队决定创建loc和iloc运算符分别用于严格处理基于标签和基于整数的索引。

ix索引依然存在，但已经弃用。我并不推荐使用它。

表5-4：DataFrame索引选项

类型	描述
df[val]	从 DataFrame 中选择单列或列序列；特殊情况的便利：布尔数组（过滤行），切片（切片行）或布尔值 DataFrame（根据某些标准设置的值）
df.loc[val]	根据标签选择 DataFrame 的单行或多行
df.loc[:, val]	根据标签选择单列或多列
df.loc[val1, val2]	同时选择行和列中的一部分
df.iloc[where]	根据整数位置选择单行或多行
df.iloc[:, where]	根据整数位置选择单列或多列
df.iloc[where_i, where_j]	根据整数位置选择行和列
df.at[label_i, label_j]	根据行、列标签选择单个标量值
df.iat[i, j]	根据行、列整数位置选择单个标量值
reindex 方法	通过标签选择行或列
get_value, set_value 方法	根据行和列的标签设置单个值

5.2.4　整数索引

在pandas对象使上用整数索引对新用户来说经常会产生歧义，这是因为它和在列表、元组等Python内建数据结构上进行索引有些许不同。例如，你可能认为下面的代码会产生错误：

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

在上面的例子中，pandas可以“回退”到整数索引，但是这样的方式难免会引起一些微小的错误。假设我们有一个索引，它包含了0、1、2，但是推断用户所需要的索引方式（标签索引或位置索引）是很难的：

```
In [144]: ser
Out[144]:
0     0.0
1     1.0
2     2.0
dtype: float64
```

另一方面，对于非整数索引，则不会有潜在的歧义：

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

为了保持一致性，如果你有一个包含整数的轴索引，数据选择时请始终使用标签索引。为了更精确地处理，可以使用loc（用于标签）或iloc（用于整数）：

```
In [147]: ser[:1]
Out[147]:
0     0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0     0.0
```

```
1    1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0    0.0
dtype: float64
```

5.2.5 算术和数据对齐

不同索引的对象之间的算术行为是pandas提供给一些应用的一项重要特性。当你将对象相加时，如果存在某个索引对不相同，则返回结果的索引将是索引对的并集。对数据库用户来说，这个特性类似于索引标签的自动外连接（outer join）。让我们看下面的示例：

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd',

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

将这些对象相加会产生：

```
In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

没有交叠的标签位置上，内部数据对齐会产生缺失值。缺失值会在后续的算术操作上产生影响。在DataFrame的示例中，行和列上都会执行对齐：

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=l
   .....:                  index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=
   .....:                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
In [157]: df1
Out[157]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
In [158]: df2
Out[158]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

将这些对象加在一起，返回一个DataFrame，它的索引、列是每个DataFrame的索引、列的并集：

```
In [159]: df1 + df2
Out[159]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

由于'c'列和'e'列并不是两个DataFrame共有的列，这两列中产生了缺失值。对于行标签不同的DataFrame对象也是如此。

如果你将两个行或列完全不同的DataFrame对象相加，结果将全部为空：

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2
```

```
In [163]: df2
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    A   B
0 NaN NaN
1 NaN NaN
```

5.2.5.1 使用填充值的算术方法

在两个不同的索引化对象之间进行算术操作时，你可能会想要使用特殊填充值，比如当轴标签在一个对象中存在，在另一个对象中不存在时，你想将缺失值填充为0：

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [169]: df2
Out[169]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

将这些df添加到一起会导致在一些不重叠的位置出现NA值：

```
In [170]: df1 + df2
Out[170]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
```

```
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

在df1上使用add方法，我将df2和一个fill_value作为参数传入：

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

表5-5中是Series和DataFrame的算术方法。这些方法中的每一个都有一个以r开头的副本，这些副本方法的参数是翻转的。因此下面两个语句的结果是等价的：

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

与此相关的一点，当对Series或DataFrame重建索引时，你也可以指定一个不同的填充值：

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

表5-5：灵活算术方法

方法	描述
add, radd	加法 (+)
sub, rsub	减法 (-)
div, rdiv	除法 (/)
floordiv, rfloordiv	整除 (//)
mul, rmul	乘法 (*)
pow, rpow	幂次方 (**)

5.2.5.2 DataFrame和Series间的操作

DataFrame和Series间的算术操作与NumPy中不同维度数组间的操作类似。首先，在下面的生动示例中，考虑二维数组和其中一行之间的区别：

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])

In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

当我们从arr中减去arr[0]时，减法在每一行都进行了操作。这就是所谓的广播机制，后续将会在附录A中涉及对通用NumPy数组的内容更深入的解释。DataFrame和Series间的操作是类似的：

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                     index=['Utah', 'Ohio', 'Texas', 'Oregon

In [180]: series = frame.iloc[0]

In [181]: frame
Out[181]:
          b    d    e
```

```
Utah     0.0   1.0   2.0
Ohio     3.0   4.0   5.0
Texas    6.0   7.0   8.0
Oregon   9.0  10.0  11.0

In [182]: series
Out[182]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

默认情况下，DataFrame和Series的数学操作中会将Series的索引和DataFrame的列进行匹配，并广播到各行：

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

如果一个索引值不在DataFrame的列中，也不在Series的索引中，则对象会重建索引并形成联合：

```
In [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [185]: frame + series2
Out[185]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

如果你想改为在列上进行广播，在行上匹配，你必须使用算术方法中的一种。例如：

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          b     d     e
Utah    0.0   1.0   2.0
```

```
Ohio     3.0   4.0   5.0
Texas    6.0   7.0   8.0
Oregon   9.0  10.0  11.0

In [188]: series3
Out[188]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

你传递的axis值是用于匹配轴的。上面的示例中表示我们需要在DataFrame的行索引上对行匹配（axis='index'或axis=0），并进行广播。

5.2.6 函数应用和映射

NumPy的通用函数（逐元素数组方法）对pandas对象也有效：

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('I
   .....:                       index=['Utah', 'Ohio', 'Texas', 'Oregon

In [191]: frame
Out[191]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

另一个常用的操作是将函数应用到一行或一列的一维数组上。DataFrame的apply方法可以实现这个功能：

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

这里的函数f，可以计算Series最大值和最小值的差，会被frame中的每一列调用一次。结果是一个以frame的列作为索引的Series。

如果你传递axis='columns'给apply函数，函数将会被每行调用一次：

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah      0.998382
```

```
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

大部分最常用的数组统计（比如sum和mean）都是DataFrame的方法，因此计算统计值时使用apply并不是必需的。

传递给apply的函数并不一定要返回一个标量值，也可以返回带有多个值的Series：

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'ma

In [197]: frame.apply(f)
Out[197]: 
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

逐元素的Python函数也可以使用。假设你想要根据frame中的每个浮点数计算一个格式化字符串，可以使用applymap方法：

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]: 
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

使用applymap作为函数名是因为Series有map方法，可以将一个逐元素的函数应用到Series上：

```
In [200]: frame['e'].map(format)
Out[200]: 
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

5.2.7 排序和排名

根据某些准则对数据集进行排序是另一个重要的内建操作。如需按行或列索引进行字典型排序，需要使用sort_index方法，该方法返回一个新的、排序好的对象：

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```

在DataFrame中，你可以在各个轴上按索引排序：

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [205]: frame.sort_index(axis=1)
Out[205]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

数据默认会升序排序，但是也可以按照降序排序：

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

如果要根据Series的值进行排序，使用sort_values方法：

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2   -3
3    2
0    4
1    7
dtype: int64
```

默认情况下，所有的缺失值都会被排序至Series的尾部：

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

当对DataFrame排序时，你可以使用一列或多列作为排序键。为了实现这个功能，传递一个或多个列名给sort_values的可选参数by：

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1

In [212]: frame
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

对多列排序时，传递列名的列表：

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

排名是指对数组从1到有效数据点总数分配名次的操作。Series和DataFrame的rank方法是实现排名的方法，默认情况下，rank通过将平均排名分配到每个组来打破平级关系：

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

排名也可以根据他们在数据中的观察顺序进行分配：

```
In [217]: obj.rank(method='first')
Out[217]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

在上面的例子中，对条目0和2设置的名次为6和7，而不是之前的平均排名6.5，是因为在数据中标签0在标签2的前面。

你可以按降序排名：

```
# 将值分配给组中的最大排名
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

表5-6是可用的平级关系打破方法列表。

DataFrame可以对行或列计算排名：

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0,
   .....:                       'c': [-2, 5, 8, -2.5]})

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

表5-6：排名中的平级关系打破方法

方法	描述
'average'	默认：在每个组中分配平均排名
'min'	对整个组使用最小排名
'max'	对整个组使用最大排名
'first'	按照值在数据中出现的次序分配排名
'dense'	类似于 method='min'，但组间排名总是增加 1，而不是一个组中的相等元素的数量

5.2.8 含有重复标签的轴索引

目前为止我们所见过的示例中，轴索引都是唯一的（索引值）。尽管很多pandas函数（比如reindex）需要标签是唯一的，但这个并不是强制性的。让我们考虑一个小型的带有重复索引的Series：

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a     0
a     1
b     2
b     3
c     4
dtype: int64
```

索引的is_unique属性可以告诉你它的标签是否唯一：

```
In [224]: obj.index.is_unique
Out[224]: False
```

带有重复索引的情况下，数据选择是与之前操作有差别的主要情况。根据一个标签索引多个条目会返回一个序列，而单个条目会返回标量值：

```
In [225]: obj['a']
Out[225]:
a     0
a     1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

这可能会使代码更复杂，因为来自索引的输出类型可能因标签是否重复而有所不同。

相同的逻辑可以拓展到在DataFrame中进行行索引：

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a',
```

```
In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

5.3　描述性统计的概述与计算

pandas对象装配了一个常用数学、统计学方法的集合。其中大部分属于归约或汇总统计的类别，这些方法从DataFrame的行或列中抽取一个Series或一系列值的单个值（如总和或平均值）。与NumPy数组中的类似方法相比，它们内建了处理缺失值的功能。考虑一个小型DataFrame：

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                     [np.nan, np.nan], [0.75, -1.3]],
   .....:                    index=['a', 'b', 'c', 'd'],
   .....:                    columns=['one', 'two'])

In [231]: df
Out[231]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

调用DataFrame的sum方法返回一个包含列上加和的Series：

```
In [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

传入axis='columns'或axis=1，则会将一行上各个列的值相加：

```
In [233]: df.sum(axis='columns')
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

除非整个切片上（在本例中是行或列）都是NA，否则NA值是被自动排除的。可以通过禁用skipna来实现不排除NA值：

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

表5-7是归约方法的常用可选参数列表。

表5-7：归约方法可选参数

方法	描述
axis	归约轴，0 为行向，1 为列向
skipna	排除缺失值，默认为 True
level	如果轴是多层索引的（MultiIndex），该参数可以缩减分组层级

一些方法，比如idxmin和idxmax，返回的是间接统计信息，比如最小值或最大值的索引值：

```
In [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

除了归约方法外，有的方法是积累型方法：

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

还有一类方法既不是归约型方法也不是积累型方法。describe就是其中之一，它一次性产生多个汇总统计：

```
In [237]: df.describe()
Out[237]:
            one       two
```

```
count   3.000000  2.000000
mean    3.083333 -2.900000
std     3.493685  2.262742
min     0.750000 -4.500000
25%     1.075000 -3.700000
50%     1.400000 -2.900000
75%     4.250000 -2.100000
max     7.100000 -1.300000
```

对于非数值型数据，describe产生另一种汇总统计：

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count     16
unique     3
top        a
freq       8
dtype: object
```

表5-8是汇总统计及其相关方法的完整列表。

表5-8：描述性统计和汇总统计

方法	描述
count	非 NA 值的个数
describe	计算 Series 或 DataFrame 各列的汇总统计集合
min, max	计算最小值、最大值
argmin, argmax	分别计算最小值、最大值所在的索引位置（整数）
idxmin, idxmax	分别计算最小值或最大值所在的索引标签
quantile	计算样本的从 0 到 1 间的分位数
sum	加和
mean	均值
median	中位数（50% 分位数）
mad	平均值的平均绝对偏差
prod	所有值的积
var	值的样本方差
std	值的样本标准差
skew	样本偏度（第三时刻）值
kurt	样本峰度（第四时刻）的值
cumsum	累计值
cummin, cummax	累计值的最小值或最大值
cumprod	值的累计积
diff	计算第一个算术差值（对时间序列有用）
pct_change	计算百分比

5.3.1　相关性和协方差

一些汇总统计，比如相关性和协方差，是由多个参数计算出的。考虑某些使用附加pandas-datareader库从Yahoo！Finance上获取的包含股价和交易量的DataFrame。如果你还没有安装它，可以通过conda或pip进行安装：

```
conda install pandas-datareader
```

为了获得一些股票行情，我使用pandas_datareader模块下载一些数据：

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
                     for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                      for ticker, data in all_data.items()})
```

当你阅读本书时，Yahoo！Finance可能已经不存在了，因为它在2017年已经被Verizon收购。请参考pandas-datareader的在线文档获取最新功能。

现在我计算股价的百分比，还有一些时间序列操作将会在第11章中深入探索：

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

Series的corr方法计算的是两个Series中重叠的、非NA的、按索引对

齐的值的相关性。相应地，cov计算的是协方差：

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

由于MSFT是一个有效的Python属性，我们可以使用更为简洁的语法来获得这些数据：

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

另一方面，DataFrame的corr和cov方法会分别以DataFrame的形式返回相关性和协方差矩阵：

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [248]: returns.cov()
Out[248]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

使用DataFrame的corrwith方法，你可以计算出DataFrame中的行或列与另一个序列或DataFrame的相关性。该方法传入一个Series时，会返回一个含有为每列计算相关性值的Series：

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
```

```
dtype: float64
```

传入一个DataFrame时，会计算匹配到列名的相关性数值。在这里，我计算出交易量百分比变化的相关性：

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

传入axis='columns'会逐行地进行计算。在所有例子中，在计算相关性之前，数据点已经按标签进行了对齐。

5.3.2　唯一值、计数和成员属性

另一类相关的方法可以从一维Series包含的数值中提取信息。为了说明这些方法，请考虑这个例子：

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', '
```

第一个函数是unique，它会给出Series中的唯一值：

```
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

唯一值并不一定按照排序好的顺序返回，但是如果需要的话可以进行排序（uniques.sort（））。相应地，value_counts计算Series包含的值的个数：

```
In [254]: obj.value_counts()
Out[254]:
c     3
a     3
b     2
d     1
dtype: int64
```

为了方便，返回的Series会按照数量降序排序。value_counts也是有效的pandas顶层方法，可以用于任意数组或序列：

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a     3
b     2
c     3
d     1
dtype: int64
```

isin执行向量化的成员属性检查，还可以将数据集以Series或

DataFrame一列的形式过滤为数据集的值子集：

```
In [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [259]: obj[mask]
Out[259]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

与isin相关的Index.get_indexer方法，可以提供一个索引数组，这个索引数组可以将可能非唯一值数组转换为另一个唯一值数组：

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

表5-9是这些方法的参考。

表5-9：唯一值、计数和集合成员属性方法

方法	描述
isin	计算表征 Series 中每个值是否包含于传入序列的布尔值数组
match	计算数组中每个值的整数索引，形成一个唯一值数组。有助于数据对齐和 join 类型的操作
unique	计算 Series 值中的唯一值数组，按照观察顺序返回
value_counts	返回一个 Series，索引是唯一值序列，值是计数个数，按照个数降序排序

某些情况下，你可能想要计算DataFrame多个相关列的直方图，如下面的例子：

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})

In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

将pandas.value_counts传入DataFrame的apply函数可以得到：

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

这里，结果中的行标签是所有列中出现的不同值，数值则是这些不同值在每个列中出现的次数。

5.4　本章小结

下一章中，我们会讨论如何利用pandas读取（或载入）和写入数据集。之后，我们会更深入地讨论使用pandas进行数据清洗、规整、分析和可视化工具。

第6章　数据载入、存储及文件格式

访问数据是使用本书中各类工具所必需的第一步。我将重点关注使用pandas进行数据输入和输出，尽管其他库中有许多工具可帮助读取和写入各种格式的数据。

输入和输出通常有以下几种类型：读取文本文件及硬盘上其他更高效的格式文件、从数据库载入数据、与网络资源进行交互（比如Web API）。

6.1 文本格式数据的读写

将表格型数据读取为DataFrame对象是pandas的重要特性。表6-1总结了部分实现该功能的函数，read_csv和read_table可能是后期我们使用最多的函数。

表6-1：Pandas的解析函数

函数	描述
read_csv	从文件、URL 或文件型对象读取分隔好的数据，逗号是默认分隔符
read_table	从文件、URL 或文件型对象读取分隔好的数据，制表符（'\t'）是默认分隔符
read_fwf	从特定宽度格式的文件中读取数据（无分隔符）
read_clipboard	read_table 的剪贴板版本，在将表格从 Web 页面上转换成数据时有用
read_excel	从 Excel 的 XLS 或 XLSX 文件中读取表格数据
read_hdf	读取用 pandas 存储的 HDF5 文件
read_html	从 HTML 文件中读取所有表格数据
read_json	从 JSON（JavaScript Object Notation）字符串中读取数据
read_msgpack	读取 MessagePack 二进制格式的 pandas 数据
read_pickle	读取以 Python pickle 格式存储的任意对象
read_sas	读取存储在 SAS 系统中定制存储格式的 SAS 数据集
read_sql	将 SQL 查询的结果（使用 SQLAlchemy）读取为 pandas 的 DataFrame
read_stata	读取 Stata 格式的数据集
read_feather	读取 Feather 二进制格式

我会解释这些函数将文本数据转换为DataFrame的机制，这些函数的可选参数主要有以下几种类型。

索引

可以将一或多个列作为返回的DataFrame，从文件或用户处获得列名，或者没有列名。

类型推断和数据转换

包括用户自定义的值转换和自定义的缺失值符号列表。

日期时间解析

包括组合功能，也包括将分散在多个列上的日期和时间信息组合成结果中的单个列。

迭代

支持对大型文件的分块迭代。

未清洗数据问题

跳过行、页脚、注释以及其他次要数据，比如使用逗号分隔千位的数字。

由于现实世界中的数据非常混乱，随着时间推移，一些数据加载函数（尤其是read_csv）的可选参数变得非常复杂。面对大量不同的参数，感到困难是正常的（read_csv在撰写本书时已经有超过50个参数了）。pandas的在线文档有大量示例展示这些参数是如何工作的。如果你在读取某个文件时遇到了困难，在文档中可能会有相似的示例帮助你找到正确的参数。

一些数据载入函数，比如pandas.read_csv，会进行类型推断，因为列的数据类型并不是数据格式的一部分。那就意味着你不必指定哪一列是数值、整数、布尔值或字符串。其他的数据格式，比如HDF5、Feather和msgpack在格式中已经存储了数据类型。

处理日期和其他自定义类型可能需要额外的努力。让我们从一个小型的逗号分隔文本文件（CSV）开始：

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

这里我使用了Unix shell的cat命令来打印文件的原始内容。如果你的操作系统是Windows，可以使用type来代替cat达到同样的效果。

由于这个文件是逗号分隔的，我们可以使用read_csv将它读入一个DataFrame：

```
In [9]: df = pd.read_csv('examples/ex1.csv')
```

```
In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

我们也可以使用read_table，并指定分隔符：

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

有的文件并不包含表头行。考虑以下文件：

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

要读取该文件，你需要选择一些选项。你可以允许pandas自动分配默认列名，也可以自己指定列名：

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd',
Out[14]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

假设你想要message列成为返回DataFrame的索引，你可以指定位置4的列为索引，或将'message'传给参数index_col：

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='mes
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

当你想要从多个列中形成一个分层索引，需要传入一个包含列序号或列名的列表：

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

在某些情况下，一张表的分隔符并不是固定的，使用空白或其他方式来分隔字段。考虑如下文本文件：

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
```

```
'ccc -0.264273 -0.386314 -0.217601\n',
'ddd -0.871858 -0.348382  1.100491\n']
```

当字段是以多种不同数量的空格分开时，尽管你可以手工处理，但在这些情况下也可以向read_table传入一个正则表达式作为分隔符。在本例中，正则表达式为\s+，因此我们可以得到：

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

本例中，由于列名的数量比数据的列数少一个，因此read_table推断第一列应当作为DataFrame的索引。

解析函数有很多附加参数帮助你处理各种发生异常的文件格式（表6-2列举了一部分）。例如，你可以使用skiprows来跳过第一行、第三行和第四行：

```
In [23]: !cat examples/ex4.csv
# 嘿!
a,b,c,d,message
# 只是为了让你觉得更难
# 谁用计算机读取CSV文件？
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

缺失值处理是文件解析过程中一个重要且常常微妙的部分。通常情况下，缺失值要么不显示（空字符串），要么用一些标识值。默认情况下，pandas使用一些常见的标识，例如NA和NULL：

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')

In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

na_values选项可以传入一个列表或一组字符串来处理缺失值：

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL']

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

在字典中，每列可以指定不同的缺失值标识：

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

表6-2列举了pandas.read_csv和pandas.read_table中常用的选项。

表6-2：一些read_csv/read_table函数参数

参数	描述
path	表明文件系统位置的字符串、URL 或文件型对象
sep 或 delimiter	用于分隔每行字段的字符序列或正则表达式
header	用作列名的行号，默认是 0（第一行），如果没有列名的话，应该为 None
index_col	用作结果中行索引的列号或列名，可以是一个单一的名称 / 数字，也可以是一个分层索引
names	结果的列名列表，和 header=None 一起用
skiprows	从文件开头处起，需要跳过的行数或行号列表
na_values	需要用 NA 替换的值序列
comment	在行结尾处分隔注释的字符
parse_dates	尝试将数据解析为 datetime，默认是 False。如果为 True，将尝试解析所有的列。也可以指定列号或列名列表来进行解析。如果列表的元素是元组或列表，将会把多个列组合在一起进行解析（例如日期 / 时间将拆分为两列）
keep_date_col	如果连接列到解析日期上，保留被连接的列，默认是 False
converters	包含列名称映射到函数的字典（例如 {'foo':f} 会把函数 f 应用到 'foo' 列）
dayfirst	解析非明确日期时，按照国际格式处理（例如 7/6/2012 -> June 7,2012），默认为 False

表6-2：一些read_csv/read_table函数参数（续）

参数	描述
date_parser	用于解析日期的函数
nrows	从文件开头处读入的行数
iterator	返回一个 TextParser 对象，用于零散地读入文件
chunksize	用于迭代的块大小
skip_footer	忽略文件尾部的行数
verbose	打印各种解析器输出的信息，比如位于非数值列中的缺失值数量
encoding	Unicode 文本编码（例如 'utf-8' 用于表示 UTF-8 编码的文本）
squeeze	如果解析数据只包含一列，返回一个 Series
thousands	千位分隔符（例如 ',' 或 '.'）

6.1.1 分块读入文本文件

当处理大型文件或找出正确的参数集来正确处理大文件时，你可能需要读入文件的一个小片段或者按小块遍历文件。

在尝试大文件之前，我们可以先对pandas的显示设置进行调整，使之更为紧凑：

```
In [33]: pd.options.display.max_rows = 10
```

现在我们可以得到：

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

如果你只想读取一小部分行（避免读取整个文件），可以指明nrows：

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

为了分块读入文件，可以指定chunksize作为每一块的行数：

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

read_csv返回的TextParser对象允许你根据chunksize遍历文件。例如，我们可以遍历ex6.csv，并对'key'列聚合获得计数值：

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

我们可以得到：

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

TextParser还具有get_chunk方法，允许你按照任意大小读取数据块。

6.1.2　将数据写入文本格式

数据可以导出为分隔的形式。让我们看下之前读取的CSV文件：

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

使用DataFrame的to_csv方法，我们可以将数据导出为逗号分隔的文件：

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

当然，其他的分隔符也是可以的（写入到sys.stdout时，控制台中打印的文本结果）：

```
In [45]: import sys

In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

缺失值在输出时以空字符串出现。你也许想要用其他标识值对缺失值进行标注：

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
```

```
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

如果没有其他选项被指定的话，行和列的标签都会被写入。不过二者也都可以禁止写入：

```
In [48]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

你也可以仅写入列的子集，并且按照你选择的顺序写入：

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series也有to_csv方法：

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

6.1.3 使用分隔格式

绝大多数的表型数据都可以使用函数pandas.read_table从硬盘中读取。然而，在某些情况下，一些手动操作可能是必不可少的。接收一个带有一行或多行错误的文件并不少见，read_table也无法解决这种情况。为了介绍一些基础工具，考虑如下的小型CSV文件：

```
In [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

对于任何带有单字符分隔符的文件，你可以使用Python的内建csv模块。要使用它，需要将任一打开的文件或文件型对象传给csv.reader：

```
import csv
f = open('examples/ex7.csv')
reader = csv.reader(f)
```

像遍历文件那样遍历reader会产生元组，元组的值为删除了引号的字符：

```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

之后，你就可以自行做一些必要处理，以将数据整理为你需要的形式。让我们按部就班，首先将文件读取为行的列表：

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

然后，我们将数据拆分为列名行和数据行：

```
In [58]: header, values = lines[0], lines[1:]
```

再然后，我们使用字典推导式和表达式zip（*values）生成一个包含数据列的字典，字典中行转置成列：

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV文件有多种不同风格。如需根据不同的分隔符、字符串引用约定或行终止符定义一种新的格式时，我们可以使用csv.Dialect定义一个简单的子类：

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

我们也可以不必定义子类，直接将CSV方言参数传入csv.reader的关键字参数：

```
reader = csv.reader(f, delimiter='|')
```

表6-3中列出了csv.Dialect中的一些属性及其用途。

表6-3：CSV方言选项

参数	描述
delimiter	一个用于分隔字段的字符，默认是 ','
lineterminator	行终止符，默认是 '\r\n'，读取器会忽略行终止符并识别跨平台行终止符
quotechar	用在含有特殊字符字段中的引号，默认是 ' " '
quoting	引用惯例。选项包括 csv.QUOTE_ALL（引用所有的字段），csv.QUOTE_MINIMAL（只使用特殊字符，如分隔符），csv.QUOTE_NONNUMERIC 和 csv.QUOTE_NONE（不引用）。细节请参考 Python 的文档。默认是 QUOTE_MINIMAL
skipinitialspace	忽略每个分隔符后的空白，默认是 False
doublequote	如何处理字段内部的引号。如果为 True，则是双引号（完整细节和行为请参考在线文档）
escapechar	当引用设置为 csv.QUOTE_NONE 时用于转义分隔符的字符串，默认是禁用的


对于具有更复杂或固定的多字符分隔符的文件，你将无法使用csv模块。在此类情况下，你将不得不使用字符串的split方法或正则表达式方法re.split进行行拆分和其他清理工作。

需要手动写入被分隔的文件时，你可以使用csv.writer。这个函数接收一个已经打开的可写入文件对象以及和csv.reader相同的CSV方言、格式选项：

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

6.1.4 JSON数据

JSON（JavaScript Object Notation的简写）已经成为Web浏览器和其他应用间通过HTTP请求发送数据的标准格式。它是一种比CSV等表格文本形式更为自由的数据形式。请看下面的例子：

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

JSON非常接近有效的Python代码，除了它的空值null和一些其他的细微差别（例如不允许列表末尾的逗号）之外。基本类型是对象（字典）、数组（列表）、字符串、数字、布尔值和空值。对象中的所有键都必须是字符串。有几个Python库用于读写JSON数据。我将在这里使用json，因为它是内置在Python标准库中的。将JSON字符串转换为Python形式时，使用json.loads方法：

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}
```

另一方面，json.dumps可以将Python对象转换回JSON：

```
In [65]: asjson = json.dumps(result)
```

你将自行决定如何将JSON对象或对象列表转换为DataFrame或其他数

据结构。比较方便的方式是将字典构成的列表（之前是JSON对象）传入DataFrame构造函数，并选出数据字段的子集：

```
In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name',

In [67]: siblings
Out[67]:
    name  age
0  Scott   30
1  Katie   38
```

pandas.read_json可以自动将JSON数据集按照指定次序转换为Series或DataFrame。例如：

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

pandas.read_json的默认选项是假设JSON数组中的每个对象是表里的一行：

```
In [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

如需了解读取、操作JSON数据（包括嵌套记录）的拓展示例，请参看第7章的USDA食品数据库示例。

如果你需要从pandas中将数据导出为JSON，一种办法是对Series和DataFrame使用to_json方法：

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

6.1.5 XML和HTML：网络抓取

Python拥有很多可以对HTML和XML格式进行读取、写入数据的库，例如lxml（http://lxml.de ）、Beautiful Soup和html5lib。尽管lxml是相对更快的库，但其他库可以更好地处理异常的HTML或XML文件。

pandas的内建函数read_html可以使用lxml和Beautiful Soup等库将HTML中的表自动解析为DataFrame对象。为了展示这个功能，我从美国FDIC政府机构下载了显示银行倒闭数据的HTML文件（在pandas文档中使用）。首先，你必须安装read_html所使用的附加库：

```
conda install lxml
pip install beautifulsoup4 html5lib
```

如果你不用conda，pip install lxml也是可以的。

pandas.read_html函数有很多选项，但是默认情况下，它会搜索并尝试解析所有包含在<table>标签中的表格型数据，返回的结果是DataFrame对象的列表：

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                      Bank Name             City  ST   CERT  \
0                   Allied Bank         Mulberry  AR     91
1  The Woodbury Banking Company         Woodbury  GA  11297
2        First CornerStone Bank  King of Prussia  PA  35312
3            Trust Company Bank          Memphis  TN   9956
4    North Milwaukee State Bank        Milwaukee  WI  20364
                Acquiring Institution        Closing Date       Updated Date
0                         Today's Bank  September 23, 2016  November 17, 2016
1                          United Bank     August 19, 2016  November 17, 2016
2  First-Citizens Bank & Trust Company         May 6, 2016  September 6, 2016
3           The Bank of Fayette County      April 29, 2016  September 6, 2016
4  First-Citizens Bank & Trust Company      March 11, 2016      June 16, 2016
```

因为failures有很多列，pandas在行内插入了换行符\。

你会在后续章节中学到，此处我们可以着手一些数据清理和分析工作，比如计算每年银行倒闭的数量：

```
In [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010    157
2009    140
2011     92
2012     51
2008     25
       ...
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, Length: 15, dtype: int64
```

6.1.5.1　使用lxml.objectify解析XML

XML（eXtensible Markup Language）是另一种常用的结构化数据格式，它使用元数据支持分层、嵌套数据。本书实际上也是从一系列大型XML文档中生成的。

之前，我展示了pandas.read_html函数，它使用lxml或Beautiful Soup从HTML中解析数据。XML和HTML结构类似，但是XML更通用。这里我将用一个例子展示如何使用lxml从更为通用的XML格式解析数据。

纽约大都会交通局（MTA）发布了一份关于其公交、火车服务（http://www.mta.info/developers/down load html ）的数据集。这里我们将关注性能数据，这部分数据包含在一个XML文件集合中。每个火车或公交服务都有一个不同的文件（例如Performance_MNR.xml代表地铁-北铁路），文件中以一系列XML记录的方式包含了按月的数据，如下面的例子：

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
```

```
  systemwide. The availability rate is based on physical observation
  the morning of regular business days only. This is a new indicator
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

使用lxml.objectify，我们可以解析这个文件，并用getroot来获得对XML文件的根节点的引用：

```
from lxml import objectify

path = 'examples/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

root.INDICATOR返回一个生成器，可以产生每一个＜INDICATOR＞XML元素。对于每条记录，我们可以将标签名称的字典（如YTD_ACTUAL）填充为数据值（不包括几个标签）：

```
data = []
skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

最后，将包含字典的列表转换为DataFrame：

```
In [81]: perf = pd.DataFrame(data)
```

```
In [82]: perf.head()
Out[82]:
Empty DataFrame
Columns: []
Index: []
```

XML数据可以比例子更复杂。每个标签也可以包含元数据。考虑一个HTML连接标签，也是有效的XML：

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

现在可以访问标签或链接文本中的任何字段（如href）：

```
In [84]: root
Out[84]: <Element a at 0x7f6b15817748>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2 二进制格式

使用Python内建的pickle序列化模块进行二进制格式操作是存储数据（也称为序列化）最高效、最方便的方式之一。pandas对象拥有一个to_pickle方法可以将数据以pickle格式写入硬盘：

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

你可以直接使用内建的pickle读取文件中“pickle化”的对象，或更方便地使用pandas.read_pickle做上述操作：

```
In [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```



pickle仅被推荐作为短期的存储格式。问题在于pickle很难确保格式的长期有效性；一个今天被pickle化的对象可能明天会因为库的新版本而无法反序列化。我们尽可能保持向后兼容性，但在将来的某个时候，可能有必要“打破”pickle格式。

pandas内建支持其他两个二进制格式：HDF5和MessagePack。我会在下一节中给出一些HDF5示例，但我推荐你尝试不同的文件格式去看看它们的速度以及它们如何用于你的分析。pandas或NumPy其他的存储格式包括：

bcolz（http://bcolz.blosc.org/ ）

基于Blosc压缩库的可压缩列式二进制格式。

Feather（http://github.com/wesm/feather ）

R编程社区的Hadley Wickham（http://hadley.nz/ ）设计的跨语言列式文件格式。Feather使用Apache箭头（http://arrow.apache.org ）列式存储器格式。

6.2.1　使用HDF5格式

HDF5是一个备受好评的文件格式，用于存储大量的科学数组数据。它以C库的形式提供，并且具有许多其他语言的接口，包括Java、Julia、MATLAB和Python。HDF5中的“HDF”代表分层数据格式。每个HDF5文件可以存储多个数据集并且支持元数据。与更简单的格式相比，HDF5支持多种压缩模式的即时压缩，使得重复模式的数据可以更高效地存储。HDF5适用于处理不适合在内存中存储的超大型数据，可以使你高效读写大型数组的一小块。

尽管可以通过使用PyTables或h5py等库直接访问HDF5文件，但pandas提供了一个高阶的接口，可以简化Series和DataFrame的存储。HDFStore类像字典一样工作并处理低级别细节：

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1            frame          (shape->[100,1])

/obj1_col        series         (shape->[100])

/obj2            frame_table    (typ->appendable,nrows->100,ncols->1,indexe
[index])
/obj3            frame_table    (typ->appendable,nrows->100,ncols->1,indexe
[index])
```

包含在HDF5文件中的对象可以使用相同的字典型API进行检索：

```
In [97]: store['obj1']
Out[97]:
            a
0   -0.204708
1    0.478943
2   -0.519439
3   -0.555730
```

```
4   1.965781
..       ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

HDFStore支持两种存储模式，'fixed'和'table'。后者速度更慢，但支持一种特殊语法的查询操作：

```
In [98]: store.put('obj2', frame, format='table')
In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [100]: store.close()
```

put是store['obj2'] = frame方法的显式版本，但允许我们设置其他选项，如存储格式。

pandas.read_hdf函数是这些工具的快捷方法：

```
In [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```


如果你处理存储在远程服务器上的数据时，比如Amazon S3或HDFS，使用其他专门为分布式存储而设计的二进制格式更为合适，比如Apache Parquet（http://parquet.apache.org ）。Parquet和其他类似的存储格式仍然在发展中，因此本书中我并没有写相关内容。

如果你是在本地处理大量数据，我推荐你尝试PyTables和h5py，看看它们是否符合你的需求。由于很多数据分析的困难在于I/O密集（而不是CPU密集），使用像HDF5这样的工具可以大大加速你的应用。

HDF5并不是数据库，它是一种适合一次写入多次读取的数据集。尽管数据可以在任何时间添加到文件中，但如果多个写入者持续写入，文件可能会损坏。

6.2.2 读取Microsoft Excel文件

pandas也支持通过ExcelFile类或pandas.read_excel函数来读取存储在Excel 2003（或更高版本）文件中的表格型数据。这些工具内部是使用附加包xlrd和openpyxl来分别读取XLS和XLSX文件的。你可能需要使用pip或conda手动安装这些工具。

使用ExcelFile时，通过将xls或xlsx的路径传入，生成一个实例：

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

存储在表中的数据可以通过pandas.read_excel读取到DataFrame中：

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

如果你读取的是含有多个表的文件，生成ExcelFile更快，但你也可以更简洁地将文件名传入pandas.read_excel：

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

如需将pandas数据写入到Excel格式中，你必须先生成一个ExcelWriter，然后使用pandas对象的to_excel方法将数据写入：

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')
```

```
In [110]: writer.save()
```

你也可以将文件路径传给to_excel，避免直接调用ExcelWriter：

```
ExcelWriter：
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3 与Web API交互

很多网站都有公开API，通过JSON或其他格式提供数据服务。有多种方式可以利用Python来访问API；我推荐的简单易用方式是使用requests包（http://docs.python-requests.orgt ）。

要获取GitHub上最新的30条关于pandas的问题，我们可以使用附加库requests发送一个HTTP GET请求：

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issue

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

Response（响应）对象的json方法将返回一个包含解析为本地Python对象的JSON的字典：

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'BUG: rank with +-inf, # 6945'
```

data中的每个元素都是一个包含GitHub问题页面上的所有数据的字典（注释除外）。我们可以将data直接传给DataFrame，并提取感兴趣的字段：

```
In [119]: issues = pd.DataFrame(data, columns=['number','title',
   .....:                                       'labels', 'state'])

In [120]: issues
Out[120]:
    number                                              title\
0    17903                            BUG:rank with +-inf, #6945
1    17902   Revert "ERP: Raise ValueError when setting sca...
2    17901   Wrong orientation of operations between DataFr...
3    17900   added 'infer' option to compression in _get_ha...
4    17898      Last day of month should group with that month
..     ...                                                 ...
```

```
25   17854       Adding an integer-location based "get" method
26   17853   BUG: adds validation for boolean keywords in D...
27   17851   BUG: duplicate indexing with embedded non-orde...
28   17850      ImportError: No module named 'pandas.plotting'
29   17846   BUG: Ignore division by 0 when merging empty d...
                                                      Labels state
0                                                         []  open
1   [{'id': 35818298, 'url': 'https://api.github.c...  open
2                                                         []  open
3                                                         []  open
4   [{'id': 76811, 'url': 'https://api.github.com/ ...  open
..                                                                 ...
25  [{'id': 35818298, 'url': 'https://api.github.c ...  open
26  [{'id': 42670965, 'url': 'https://api.github.c ...  open
27  [{'id': 76811, 'url': 'https://api.github.com/ ...  open
28  [{'id': 31932467, 'url': 'https://api.github.c ...  open
29  [{'id': 76865106, 'url': 'https://api.github.c ...  open
[30 rows x 4 columns]
```

通过一些复杂操作，你可以创建一些更高阶的接口来访问常用的Web API，以返回DataFrame对象以便于分析。

6.4 与数据库交互

在业务场景中，大部分数据并不是储存在文本或Excel文件中的。基于SQL的关系型数据库（例如SQL Server、PostgreSQL和MySQL）使用广泛，很多小众数据库也变得越发流行。数据库的选择通常取决于性能、数据完整性以及应用的可伸缩性需求。

从SQL中将数据读取为DataFrame是相当简单直接的，pandas有多个函数可以简化这个过程。作为例子，我将使用Python内建的sqlite3驱动来生成一个SQLite数据库：

```
In [121]: import sqlite3

In [122]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

再插入几行数据：

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5)]

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>

In [129]: con.commit()
```

当从数据库的表中选择数据时，大部分Python的SQL驱动（PyODBC、psycopg2、MySQLdb、pymssql等）返回的是元组的列表：

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

你可以将元组的列表传给DataFrame构造函数，但你还需要包含在游标的description属性中的列名：

```
In [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.descripti
Out[134]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

由于内容较多，你肯定不想每次查询数据库都重复同样多的步骤。SQLAlchemy项目（http://www.sqlalchemy.org/ ）是一个流行的Python SQL工具包，抽象去除了SQL数据库之间的许多常见差异。pandas有一个read_sql函数允许你从通用的SQLAlchemy连接中轻松地读取数据。这里，我将使用SQLAlchemy连接到相同的SQLite数据库，并从之前创建的表中读取数据：

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [137]: pd.read_sql('select * from test', db)
Out[137]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5　本章小结

访问数据通常是数据分析过程的第一步。我们在本章已经学习了一些有用的工具，可以帮助你入门。在后续章节中，我们将深入数据处理、数据可视化、时间序列分析和其他主题。

第7章　数据清洗与准备

在进行数据分析和建模的过程中，大量的时间花在数据准备上：加载、清理、转换和重新排列。这样的工作占用了分析师80%以上的时间。有时，数据存储在文件或数据库中的方式对于特定的任务来说格式并不正确。许多研究人员选择使用通用编程语言（如Python、Perl、R或Java）或Unix文本处理工具（如sed或awk）进行从一种形式到另一种形式的特殊数据处理。幸运的是，pandas以及内置的Python语言功能为你提供了一个高级、灵活和快速的工具集，使你能够将数据处理为正确的形式。

如果你发现某种数据处理即不在本书的范围之内，也不在pandas库中，请随时在Python邮件列表或pandas的GitHub页面上分享你的案例。事实上，pandas中的大部分设计和实现都是由真实世界的应用需求所驱动的。

在本章中，我将讨论用于缺失值、重复值、字符串操作和其他分析数据转换的工具。下一章中，我将重点关注利用各种方式对数据集联合、重排列。

7.1　处理缺失值

缺失数据会在很多数据分析应用中出现。pandas的目标之一就是尽可能无痛地处理缺失值。例如，pandas对象的所有描述性统计信息默认情况下是排除缺失值的。

pandas对象中表现缺失值的方式并不完美，但是它对大部分用户来说是有用的。对于数值型数据，pandas使用浮点值NaN（Not a Number来表示缺失值）。我们称NaN为容易检测到的标识值：

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'a

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

在pandas中，我们采用了R语言中的编程惯例，将缺失值成为NA，意思是not available（不可用）。在统计学应用中，NA数据可以是不存在的数据或者是存在但不可观察的数据（例如在数据收集过程中出现了问题）。当清洗数据用于分析时，对缺失数据本身进行分析以确定数据收集问题或数据丢失导致的数据偏差通常很重要。

Python内建的None值在对象数组中也被当作NA处理：

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
1    False
2     True
3    False
```

```
dtype: bool
```

pandas项目持续改善处理缺失值的内部细节，但是用户API函数，比如pandas.isnull，抽象掉了很多令人厌烦的细节。表7-1是处理缺失值的相关函数列表。

表7-1：NA处理方法

函数名	描述
dropna	根据每个标签的值是否是缺失数据来筛选轴标签，并根据允许丢失的数据量来确定阈值
fillna	用某些值填充缺失的数据或使用插值方法（如 'ffill' 或 'bfill'）。
isnull	返回表明哪些值是缺失值的布尔值
notnull	isnull 的反函数

7.1.1 过滤缺失值

有多种过滤缺失值的方法。虽然你可以使用pandas.isnull和布尔值索引手动地过滤缺失值，但dropna在过滤缺失值时是非常有用的。在Series上使用dropna，它会返回Series中所有的非空数据及其索引值：

```
In [15]: from numpy import nan as NA
In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

上面的例子与下面的代码是等价的：

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

当处理DataFrame对象时，事情会稍微更复杂一点。你可能想要删除全部为NA或包含有NA的行或列。dropna默认情况下会删除包含缺失值的行：

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [22]: cleaned
```

```
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

传入how='all'时，将删除所有值均为NA的行：

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

如果要用同样的方式去删除列，传入参数axis=1：

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

过滤DataFrame的行的相关方法往往涉及时间序列数据。假设你只想保留包含一定数量的观察值的行。你可以用thresh参数来表示：

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
```

```
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

7.1.2 补全缺失值

你有时可能需要以多种方式补全“漏洞”，而不是过滤缺失值（也可能丢弃其他数据）。大多数情况下，主要使用fillna方法来补全缺失值。调用fillna时，可以使用一个常数来替代缺失值：

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

在调用fillna时使用字典，你可以为不同列设定不同的填充值：

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

fillna返回的是一个新的对象，但你也可以修改已经存在的对象：

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

用于重建索引的相同的插值方法也可以用于fillna：

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN

In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232

In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

使用fillna你可以完成很多带有一点创造性的工作。例如，你可以将Series的平均值或中位数用于填充缺失值：

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
```

```
3    3.833333
4    7.000000
dtype: float64
```

表7-2是fillna的参考。

表7-2：fillna函数参数

参数	描述
value	标量值或字典型对象用于填充缺失值
method	插值方法，如果没有其他参数，默认是'ffill'
axis	需要填充的轴，默认axis=0
inplace	修改被调用的对象，而不是生成一个备份
limit	用于前向或后向填充时最大的填充范围

7.2 数据转换

目前为止，我们主要讲解了数据的重新排列。过滤、清洗以及其他转换是另外一系列重要的操作。

7.2.1 删除重复值

由于各种原因，DataFrame中会出现重复行。请看如下例子：

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})

In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

DataFrame的duplicated方法返回的是一个布尔值Series，这个Series反映的是每一行是否存在重复（与之前出现过的行相同）情况：

```
In [47]: data.duplicated()
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

drop_duplicates返回的是DataFrame，内容是duplicated返回数组中为False的部分：

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

这些方法默认都是对列进行操作。你可以指定数据的任何子集来检测是否有重复。假设我们有一个额外的列，并想基于'k1'列去除重复值：

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

duplicated和drop_duplicates默认都是保留第一个观测到的值。传入参数keep='last'将会返回最后一个：

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

7.2.2　使用函数或映射进行数据转换

对于许多数据集，你可能希望基于DataFrame中的数组、列或列中的数值进行一些转换。考虑下面这些收集到的关于肉类的假设数据：

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon
   ....:                               'Pastrami', 'corned beef', 'Bacon
   ....:                              'pastrami', 'honey ham', 'nova lo
   ....:                      'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]}

In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

假设你想要添加一列用于表明每种食物的动物肉类型。让我们先写下一个食物和肉类的映射：

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

Series的map方法接收一个函数或一个包含映射关系的字典型对象，但是这里我们有一个小的问题在于一些肉类大写了，而另一部分肉类没有。因此，我们需要使用Series的str.lower方法将每个值都转换为小写：

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
```

```
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

我们也可以传入一个能够完成所有工作的函数：

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

使用map是一种可以便捷执行按元素转换及其他清洗相关操作的方法。

7.2.3　替代值

使用fillna填充缺失值是通用值替换的特殊案例。前面你已经看到，map可以用来修改一个对象中的子集的值，但是replace提供了更为简单灵活的实现。让我们考虑下面的Series：

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0        1.0
1     -999.0
2        2.0
3     -999.0
4    -1000.0
5        3.0
dtype: float64
```

-999可能是缺失值的标识。如果要使用NA来替代这些值，我们可以使用replace方法生成新的Series（除非你传入了inplace=True）：

```
In [62]: data.replace(-999, np.nan)
Out[62]:
0        1.0
1        NaN
2        2.0
3        NaN
4    -1000.0
5        3.0
dtype: float64
```

如果你想要一次替代多个值，你可以传入一个列表和替代值：

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
0     1.0
1     NaN
2     2.0
3     NaN
4     NaN
5     3.0
dtype: float64
```

要将不同的值替换为不同的值，可以传入替代值的列表：

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

参数也可以通过字典传递：

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

 data.replace方法与data.str.replace方法是不同的，data.str.replace是对字符串进行按元素替代的。我们将在下一章看到Series的字符串方法。

7.2.4 重命名轴索引

和Series中的值一样，可以通过函数或某种形式的映射对轴标签进行类似的转换，生成新的且带有不同标签的对象。你也可以在不生成新的数据结构的情况下修改轴。下面是简单的示例：

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=['Ohio', 'Colorado', 'New York'],
   ....:                     columns=['one', 'two', 'three', 'four'])
```

与Series类似，轴索引也有一个map方法：

```
In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

你可以赋值给index，修改DataFrame：

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

如果你想要创建数据集转换后的版本，并且不修改原有的数据集，一个有用的方法是rename：

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

值得注意的是，rename可以结合字典型对象使用，为轴标签的子集提

供新的值：

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

rename可以让你从手动复制DataFrame并为其分配索引和列属性的烦琐工作中解放出来。如果你想要修改原有的数据集，传入inplace=True：

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [74]: data
Out[74]:
         one  two  three  four
INDIANA    0    1      2     3
COLO       4    5      6     7
NEW        8    9     10    11
```

7.2.5 离散化和分箱

连续值经常需要离散化，或者分离成”箱子“进行分析。假设你有某项研究中一组人群的数据，你想将他们进行分组，放入离散的年龄框中：

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

让我们将这些年龄分为18～25、26～35、36～60以及61及以上等若干组。为了实现这个，你可以使用pandas中的cut：

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)
In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (6
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (
```

pandas返回的对象是一个特殊的Categorical对象。你看到的输出描述了由pandas.cut计算出的箱。你可以将它当作一个表示箱名的字符串数组；它在内部包含一个categories（类别）数组，它指定了不同的类别名称以及codes属性中的ages（年龄）数据标签：

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

请注意，pd.value_counts（cats）是对pandas.cut的结果中的箱数量的计数。

与区间的数学符号一致，小括号表示边是开放的，中括号表示它是封闭的（包括边）。你可以通过传递right＝False来改变哪一边是封闭的：

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [6
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [
```

你也可以通过向labels选项传递一个列表或数组来传入自定义的箱名：

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior

In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, Mi
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

如果你传给cut整数个的箱来代替显式的箱边，pandas将根据数据中的最小值和最大值计算出等长的箱。请考虑一些均匀分布的数据被切成四份的情况：

```
In [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0
(0.76, 0.97]]
```

precision＝2的选项将十进制精度限制在两位。

qcut是一个与分箱密切相关的函数，它基于样本分位数进行分箱。取决于数据的分布，使用cut通常不会使每个箱具有相同数据量的数据点。

由于qcut使用样本的分位数，你可以通过qcut获得等长的箱：

```
In [87]: data = np.random.randn(1000)  # 正态分布

In [88]: cats = pd.qcut(data, 4)  # 切成四份

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.9
 -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265
 0.62] <
                                  (0.62, 3.928]]

In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928]       250
(-0.0265, 0.62]     250
(-0.68, -0.0265]    250
(-2.95, -0.68]      250
dtype: int64
```

与cut类似，你可以传入自定义的分位数（0和1之间的数据，包括边）：

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.2
5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187
1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.02
5, 1.286] <
                                  (1.286, 3.928]]
```

后续章节中，在讨论聚合和分组操作时，我们将会继续讨论cut和qcut，因为这些离散化函数对于分位数和分组分析特别有用。

7.2.6 检测和过滤异常值

过滤或转换异常值在很大程度上是应用数组操作的事情。考虑一个具有正态分布数据的DataFrame：

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.049091     0.026112    -0.002544    -0.051827
std       0.996947     1.007458     0.995232     0.998311
min      -3.645860    -3.184377    -3.745356    -3.428254
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.525865     2.735527     3.366626
```

假设你想要找出一列中绝对值大于三的值：

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
41    -3.399312
136   -3.745356
Name: 2, dtype: float64
```

要选出所有值大于3或小于-3的行，你可以对布尔值DataFrame使用any方法：

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
```

```
803 -3.645860  0.255475 -0.549574 -1.907459
```

值可以根据这些标准来设置，下面代码限制了-3到3之间的数值：

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3
In [98]: data.describe()
Out[98]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.000000     2.735527     3.000000
```

语句np.sign（data）根据数据中的值的正负分别生成1和-1的数值：

```
In [99]: np.sign(data).head()
Out[99]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

7.2.7 置换和随机抽样

使用numpy.random.permutation对DataFrame中的Series或行进行置换（随机重排序）是非常方便的。在调用permutation时根据你想要的轴长度可以产生一个表示新顺序的整数数组：

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

整数数组可以用在基于iloc的索引或等价的take函数中：

```
In [103]: df
Out[103]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

要选出一个不含有替代值的随机子集，你可以使用Series和DataFrame的sample方法：

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

要生成一个带有替代值的样本（允许有重复选择），将replace=True传入sample方法：

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)

In [108]: draws
Out[108]:
4    4
1    7
4    4
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
dtype: int64
```

7.2.8 计算指标/虚拟变量

将分类变量转换为“虚拟”或“指标”矩阵是另一种用于统计建模或机器学习的转换操作。如果DataFrame中的一列有k个不同的值，则可以衍生一个k列的值为1和0的矩阵或DataFrame。pandas有一个get_dummies函数用于实现该功能，尽管你自行实现也不难。让我们回顾一下之前的一个示例DataFrame：

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                    'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

在某些情况下，你可能想在指标DataFrame的列上加入前缀，然后与其他数据合并。在get_dummies方法中有一个前缀参数用于实现该功能：

```
In [111]: dummies = pd.get_dummies(df['key'], prefix='key')

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

如果DataFrame中的一行属于多个类别，则情况略为复杂。让我们看看MovieLens的1M数据集，在第14章中有更为详细的介绍：

```
In [114]: mnames = ['movie_id', 'title', 'genres']
```

```
In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep
   .....:                         header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
   movie_id                               title
0         1                    Toy Story (1995)    Animation|Children's|C
1         2                      Jumanji (1995)    Adventure|Children's|F
2         3             Grumpier Old Men (1995)                Comedy|Ror
3         4            Waiting to Exhale (1995)                  Comedy|D
4         5  Father of the Bride Part II (1995)
5         6                         Heat (1995)        Action|Crime|Thril
6         7                      Sabrina (1995)              Comedy|Roma
7         8                 Tom and Huck (1995)         Adventure|Childre
8         9                 Sudden Death (1995)                        Ac
9        10                    GoldenEye (1995)    Action|Adventure|Thril
```

为每个电影流派添加指标变量需要进行一些数据处理。首先，我们从数据集中提取出所有不同的流派的列表：

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
```

然后我们得到：

```
In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-No
       'Western'], dtype=object)
```

使用全0的DataFrame是构建指标DataFrame的一种方式：

```
In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

现在，遍历每一部电影，将dummies每一行的条目设置为1。为了实

现该功能，我们使用dummies.columns来计算每一个流派的列指标：

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

之后，使用.loc根据这些指标来设置值：

```
In [126]: for i, gen in enumerate(movies.genres):
   .....:     indices = dummies.columns.get_indexer(gen.split('|'))
   .....:     dummies.iloc[i, indices] = 1
   .....:
```

之后，和前面一样，你可以将结果与movies进行联合：

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [128]: movies_windic.iloc[0]
Out[128]:
movie_id                                       1
title                           Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                1
Genre_Children's                               1
Genre_Comedy                                   1
Genre_Adventure                                0
Genre_Fantasy                                  0
Genre_Romance                                  0
Genre_Drama                                    0
                                ...
Genre_Crime                                    0
Genre_Thriller                                 0
Genre_Horror                                   0
Genre_Sci-Fi                                   0
Genre_Documentary                              0
Genre_War                                      0
Genre_Musical                                  0
Genre_Mystery                                  0
Genre_Film-Noir                                0
Genre_Western                                  0
Name: 0, Length: 21, dtype: object
```

对于更大的数据，上面这种使用多成员构建指标变量并不是特别快速。更好的方法是写一个直接将数据写为NumPy数组的底层函数，然后将结果封装进DataFrame。

将get_dummies与cut等离散化函数结合使用是统计应用的一个有用方法：

```
In [129]: np.random.seed(12345)

In [130]: values = np.random.rand(10)

In [131]: values
Out[131]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,
        0.6532,  0.7489,  0.6536])

In [132]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [133]: pd.get_dummies(pd.cut(values, bins))
Out[133]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           1
1           0           1           0           0           0
2           1           0           0           0           0
3           0           1           0           0           0
4           0           0           1           0           0
5           0           0           1           0           0
6           0           0           0           0           1
7           0           0           0           1           0
8           0           0           0           1           0
9           0           0           0           1           0
```

我们使用numpy.random.seed来设置随机种子以确保示例的确定性。我们将在本书后面的内容中再次讨论pandas.get_dummies。

7.3 字符串操作

由于Python在字符串和文本操作上的便利性，使Python成为一个流行的原生数据集操作语言已经有很长时间了。字符串对象的内建方法使得大部分文本操作非常简单。对于更为复杂的模式匹配和文本操作，正则表达式可能是需要的。pandas允许你将字符串和正则表达式简洁地应用到整个数据数组上，此外还能处理数据缺失带来的困扰。

7.3.1　字符串对象方法

在很多字符串处理和脚本应用中，内建的字符串方法是足够的。例如，一个逗号分隔的字符串可以使用split方法拆分成多块：

```
In [134]: val = 'a,b,  guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', '  guido']
```

split常和strip一起使用，用于清除空格（包括换行）：

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

这些子字符串可以使用加法与两个冒号分隔符连接在一起：

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
Out[139]: 'a::b::guido'
```

但是这并不是一个实用的通用方法。在字符串'：：'的join方法中传入一个列表或元组是一种更快且更加Pythonic（Python风格化）的方法：

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

其他方法涉及定位子字符串。使用Python的in关键字是检测子字符串的最佳方法，尽管index和find也能实现同样的功能：

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1
```

```
In [143]: val.find(':')
Out[143]: -1
```

请注意find和index的区别在于index在字符串没有找到时会抛出一个异常（而find是返回-1）：

```
In [144]: val.index(':')
---------------------------------------------------------------
ValueError                          Traceback (most recent call last)
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

相关地，count返回的是某个特定的子字符串在字符串中出现的次数：

```
In [145]: val.count(',')
Out[145]: 2
```

replace将用一种模式替代另一种模式。它通常也用于传入空字符串来删除某个模式。

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b::  guido'

In [147]: val.replace(',', '')
Out[147]: 'ab  guido'
```

表7-3是一些Python字符串方法的列表。

后续你将看到正则表达式也可以用于这些方法。

表7-3：Python内建字符串方法

方法	描述
count	返回子字符串在字符串中的非重叠出现次数
endswith	如果字符串以后缀结尾则返回 True
startswith	如果字符串以前缀开始则返回 True
join	使用字符串作为间隔符，用于粘合其他字符串的序列
index	如果在字符串中找到，则返回子字符串中第一个字符的位置；如果找不到则引发 ValueError
find	返回字符串中第一个出现子字符的第一个字符的位置；类似 index，但如果没有找到则返回 -1
rfind	返回子字符串在字符串中最后一次出现时第一个字符的位置；如果没有找到，则返回 -1
replace	使用一个字符串替代另一个字符串
strip,rstrip,lstrip	修剪空白，包括换行符；相当于对每个元素进行 x.strip()（以及 rstrip，lstrip）。
split	使用分隔符将字符串拆分为子字符串的列表
lower	将大写字母转换为小写字母
upper	将小写字母转换为大写字母
casefold	将字符转换为小写，并将任何特定于区域的变量字符组合转换为常见的可比较形式
ljust,rjust	左对齐或右对齐；用空格（或其他一些字符）填充字符串的相反侧以返回具有最小宽度的字符串

7.3.2 正则表达式

正则表达式提供了一种在文本中灵活查找或匹配（通常更为复杂的）字符串模式的方法。单个表达式通常被称为regex，是根据正则表达式语言形成的字符串。Python内建的re模块是用于将正则表达式应用到字符串上的库。我在此处会给出一些re模块的示例。

 编写正则表达式的艺术是可以自成一章的，因此不在本书的讨论范围之内。在网上和其他书中有很多优秀的教程。

re模块主要有三个主题：模式匹配、替代、拆分。当然，这三部分主题是相关联的。一个正则表达式描述了在文本中需要定位的一种模式，可以用于多种目标。让我们来看一个简单的示例：假设我们想将含有多种空白字符（制表符、空格、换行符）的字符串拆分开。描述一个或多个空白字符的正则表达式是\s+：

```
In [148]: import re

In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

当你调用re.split（'\s+'，text），正则表达式首先会被编译，然后正则表达式的split方法在传入文本上被调用。你可以使用re.compile自行编译，形成一个可复用的正则表达式对象：

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

如果你想获得的是一个所有匹配正则表达式的模式的列表，你可以使用findall方法：

```
In [153]: regex.findall(text)
Out[153]: ['    ', '\t ', '  \t']
```

 为了在正则表达式中避免转义符\的影响，可以使用原生字符串语法，比如r'C：\x'或者用等价的'C：\\x'

如果你需要将相同的表达式应用到多个字符串上，推荐使用re.compile创建一个正则表达式对象，这样做有利于节约CPU周期。

match和search与findall相关性很大。findall返回的是字符串中所有的匹配项，而search返回的仅仅是第一个匹配项。match更为严格，它只在字符串的起始位置进行匹配。作为一个不重要的示例，我们考虑下一段文本以及一个可以识别大部分电子邮件地址的正则表达式：

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
# re.IGNORECASE使正则表达式不区分大小写
regex = re.compile(pattern, flags=re.IGNORECASE)
```

在文本上使用findall会生成一个电子邮件地址的列表：

```
In [155]: regex.findall(text)
Out[155]:
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

search返回的是文本中第一个匹配到的电子邮件地址。对于前面提到的正则表达式，匹配对象只能告诉我们模式在字符串中起始和结束的位置：

```
In [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.co

In [158]: text[m.start():m.end()]
Out[158]: 'dave@google.com'
```

regex.match只在模式出现于字符串起始位置时进行匹配，如果没有

匹配到，返回None：

```
In [159]: print(regex.match(text))
None
```

相关地，sub会返回一个新的字符串，原字符串中的模式会被一个新的字符串替代：

```
In [160]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

假设您想查找电子邮件地址，并将每个地址分为三个部分：用户名，域名和域名后缀。要实现这一点，可以用括号将模式包起来：

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [162]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

由这个修改后的正则表达式产生的匹配对象的groups方法，返回的是模式组件的元组：

```
In [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

当模式可以分组时，findall返回的是包含元组的列表：

```
In [165]: regex.findall(text)
Out[165]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

sub也可以使用特殊符号，如\1和\2，访问每个匹配对象中的分组。

符号\1代表的是第一个匹配分组，\2代表的是第二个匹配分组，以此类推：

```
In [166]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', te
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Python中关于正则表达式还有更多的内容，但是大部分超过了本书的范围。表7-4提供了一个简要概述。

表7-4：正则表达式方法

方法	描述
findall	将字符串中所有的非重叠匹配模式以列表形式返回
finditer	与 findall 类似，但返回的是迭代器
match	在字符串起始位置匹配模式，也可以将模式组建匹配到分组中；如果模式匹配上了，返回的一个匹配对象，否则返回 None
search	扫描字符串的匹配模式，如果扫描到了返回匹配对象，与 match 方法不同的是，search 方法的匹配可以是字符串的任意位置，而不仅仅是字符串的起始位置
split	根据模式，将字符串拆分为多个部分
sub, subn	用替换表达式替换字符串中所有的匹配（sub）或第 n 个出现的匹配串（subn）；使用符号 \ 1、\ 2 ...... 来引用替换字符串中的匹配组元素

7.3.3 pandas中的向量化字符串函数

清理杂乱的数据集用于分析通常需要大量的字符串处理和正则化。包含字符串的列有时会含有缺失数据，使事情变得复杂：

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.co
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [168]: data = pd.Series(data)

In [169]: data
Out[169]:
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object

In [170]: data.isnull()
Out[170]:
Dave     False
Rob      False
Steve    False
Wes       True
dtype: bool
```

你可以使用data.map将字符串和有效的正则表达式方法（以lambda或其他函数的方式传递）应用到每个值上，但是在NA（null）值上会失败。为了解决这个问题，Series有面向数组的方法用于跳过NA值的字符串操作。这些方法通过Series的str属性进行调用，例如，我们可以通过str.contains来检查每个电子邮件地址是否含有'gmail'：

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

正则表达式也可以结合任意的re模块选项使用，例如IGNORECASE：

```
In [172]: pattern
```

```
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave     [(dave, google, com)]
Rob        [(rob, gmail, com)]
Steve    [(steve, gmail, com)]
Wes                        NaN
dtype: object
```

有多种方法可以进行向量化的元素检索。可以使用str.get或在str属性内部索引：

```
In [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave     True
Rob      True
Steve    True
Wes       NaN
dtype: object
```

要访问嵌入式列表中的元素，我们可以将索引传递给这些函数中的任意一个：

```
In [176]: matches.str.get(1)
Out[176]:
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
dtype: float64

In [177]: matches.str[0]
Out[177]:
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
dtype: float64
```

你可以使用字符串切片的类似语法进行向量化切片：

```
In [178]: data.str[:5]
```

```
Out[178]:
Dave       dave@
Rob        rob@g
Steve      steve
Wes          NaN
dtype: object
```

表7-5中有更多pandas字符串方法。

表7-5：部分向量化字符串方法列表

方法	描述
cat	根据可选的分隔符按元素黏合字符串
contains	返回是否含有某个模式 / 正则表达式的布尔值数组
count	模式出现次数的计数
extract	使用正则表达式从字符串 Series 中分组抽取一个或多个字符串；返回的结果是每个分组形成一列的 DataFrame
endswith	等价于对每个元素使用 x.endwith（模式）

表7-5：部分向量化字符串方法列表（续）

方法	描述
startswith	等价于对每个元素使用 x.startwith（模式）
findall	找出字符串中所有的模式 / 正则表达式匹配项，以列表返回
get	对每个元素进行索引（获得第 *i* 个元素）
isalnum	等价于内建的 str.alnum
isalhpa	等价于内建的 str.isalpha
isdecimal	等价于内建的 str.isdecimal
isdigit	等价于内建的 str.isdigit
islower	等价于内建的 str.islower
isnumeric	等价于内建的 str. isnumeric
isupper	等价于内建的 str. isupper
join	根据传递的分隔符，将 Series 中的字符串联合
len	计算每个字符串的长度
lower, upper	转换大小写；等价于对每个元素进行 x.lower() 或 x.upper()
match	使用 re.match 将正则表达式应用到每个元素上，将匹配分组以列表形式返回
pad	将空白加到字符串的左边、右边或两边
center	等价于 pad（side='both'）
repeat	重复值（例如 s.str.repeat（3）等价于对每个字符串进行 x*3）
replace	以其他字符串替代模式 / 正式表达式的匹配项
slice	对 Series 中的字符串进行切片
split	以分隔符或正则表达式对字符串进行拆分
strip	对字符串两侧的空白进行消除，包括换行符
rstrip	消除字符串右边的空白
lstrip	消除字符串左边的空白

7.4　本章小结

高效的数据准备工作使你可以将更多的时间用于分析而不是准备数据，从而显著提升生产效率。本章中，我们已经探索了多个工具，但覆盖范围仍不够全面。下一章，我们将探索pandas的数据连接和分组功能。

第8章　数据规整：连接、联合与重塑

在很多应用中，数据可能分布在多个文件或数据库中，抑或以某种不易于分析的格式进行排列。本章关注于对数据联合、连接以及重排列有用的工具。

首先，我介绍了pandas中的分层索引的概念，这个概念在这些操作中被广泛使用。然后我会深入介绍特定的数据操作。在第14章中你可以看到这些工具的各种应用方法。

8.1　分层索引

分层索引是pandas的重要特性，允许你在一个轴向上拥有多个（两个或两个以上）索引层级。笼统地说，分层索引提供了一种在更低维度的形式中处理更高维度数据的方式。让我们以一个简单的例子开始，先创建一个Series，以列表的列表（或数组）作为索引：

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

你看到的是一个以MultiIndex作为索引的Series的美化视图。索引中的“间隙”表示“直接使用上面的标签”：

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1
```

通过分层索引对象，也可以称为部分索引，允许你简洁地选择出数据的子集：

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64

In [13]: data['b':'c']
Out[13]:
```

```
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

在“内部”层级中进行选择也是可以的：

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

分层索引在重塑数据和数组透视表等分组操作中扮演了重要角色。例如，你可以使用unstack方法将数据在DataFrame中重新排列：

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

unstack的反操作是stack：

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
```

```
   3    0.769023
dtype: float64
```

stack和unstack的细节将在本章后续部分进行探索。

在DataFrame中，每个轴都可以拥有分层索引：

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....:                      index=[['a', 'a', 'b', 'b'], [1, 2, 1,
   ....:                      columns=[['Ohio', 'Ohio', 'Colorado'],
   ....:                               ['Green', 'Red', 'Green']])

In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

分层的层级可以有名称（可以是字符串或Python对象）。如果层级有名称，这些名称会在控制台输出中显示：

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

 请注意区分行标签中的索引名称'state'和'color'。

通过部分列索引，你可以选出列中的组：

```
In [23]: frame['Ohio']
Out[23]:
```

```
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

一个MultiIndex对象可以使用其自身的构造函数创建并复用。前面介绍的带有层级名称的DataFrame的列可以这样创建：

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red
                       names=['state', 'color'])
```

8.1.1 重排序和层级排序

有时，你需要重新排列轴上的层级顺序，或者按照特定层级的值对数据进行排序。swaplevel接收两个层级序号或层级名称，返回一个进行了层级变更的新对象（但是数据是不变的）：

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state       Ohio      Colorado
color      Green Red     Green
key2 key1
1    a         0   1         2
2    a         3   4         5
1    b         6   7         8
2    b         9  10        11
```

另一方面，sort_index只能在单一层级上对数据进行排序。在进行层级变换时，使用sort_index以使得结果按照层级进行字典排序也很常见：

```
In [25]: frame.sort_index(level=1)
Out[25]:
state       Ohio      Colorado
color      Green Red     Green
key1 key2
a    1         0   1         2
b    1         6   7         8
a    2         3   4         5
b    2         9  10        11

In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state       Ohio      Colorado
color      Green Red     Green
key2 key1
1    a         0   1         2
     b         6   7         8
2    a         3   4         5
     b         9  10        11
```

如果索引按照字典顺序从最外层开始排序，那么数据选择性能会更好——调用sort_index（level=0）或sort_index可以得到这样的结果。

8.1.2　按层级进行汇总统计

DataFrame和Series中很多描述性和汇总性统计有一个level选项，通过level选项你可以指定你想要在某个特定的轴上进行聚合。考虑上述示例中的DataFrame，我们可以按照层级在行或列上像下面这样进行聚合：

```
In [27]: frame.sum(level='key2')
Out[27]:
state   Ohio      Colorado
color Green Red      Green
key2
1         6   8         10
2        12  14         16

In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

上面的例子使用了pandas的groupby机制，这个机制的细节将会在本书后续章节讨论。

8.1.3　使用DataFrame的列进行索引

通常我们不会使用DataFrame中一个或多个列作为行索引；反而你可能想要将行索引移动到DataFrame的列中。下面是一个示例DataFrame：

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
   ....:                       'c': ['one', 'one', 'one', 'two', 'two',
   ....:                             'two', 'two'],
   ....:                       'd': [0, 1, 2, 0, 1, 2, 3]})

In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

DataFrame的set_index函数会生成一个新的DataFrame，新的DataFrame使用一个或多个列作为索引：

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

默认情况下这些列会从DataFrame中移除，你也可以将它们留在DataFrame中：

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
```

```
         a  b    c  d
c   d
one 0    0  7  one  0
    1    1  6  one  1
    2    2  5  one  2
two 0    3  4  two  0
    1    4  3  two  1
    2    5  2  two  2
    3    6  1  two  3
```

另一方面，reset_index是set_index的反操作，分层索引的索引层级会被移动到列中：

```
In [34]: frame2.reset_index()
Out[34]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

8.2 联合与合并数据集

包含在pandas对象的数据可以通过多种方式联合在一起：

·pandas.merge根据一个或多个键将行进行连接。对于SQL或其他关系型数据库的用户来说，这种方式比较熟悉，它实现的是数据库的连接操作。

·pandas.concat使对象在轴向上进行黏合或“堆叠”。

·combine_first实例方法允许将重叠的数据拼接在一起，以使用一个对象中的值填充另一个对象中的缺失值。

我会介绍上面几种方法，并给出一些例子。这些例子将会用在本书的后续内容中。

8.2.1 数据库风格的DataFrame连接

合并或连接操作通过一个或多个键连接行来联合数据集。这些操作是关系型数据库的核心内容（例如基于SQL的数据库）。pandas中的merge函数主要用于将各种join操作算法运用在你的数据上：

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

这是一个多对一连接的例子；df1的数据有多个行的标签为a和b，而df2在key列中每个值仅有一行。调用merge处理我们获得的对象：

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

请注意，我并没有指定在哪一列上进行连接。如果连接的键信息没有

指定，merge会自动将重叠列名作为连接的键。但是，显式地指定连接键才是好的实现：

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

如果每个对象的列名是不同的，你可以分别为它们指定列名：

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a',
   ....:                    'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                    'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

你可能注意到结果中缺少'c'和'd'的值以及相关的数据。默认情况下，merge做的是内连接（'inner'join），结果中的键是两张表的交集。其他可选的选项有'left'、'right'和'outer'。外连接（outer join）是键的并集，联合了左连接和右连接的效果：

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
```

```
7     NaN    d     2.0
```

表8-1是对how选项的总结。

表8-1：how参数的不同连接类型

选项	行为
'inner'	只对两张表都有的键的交集进行联合
'left'	对所有左表的键进行联合
'right'	对所有右表的键进行联合
'outer'	对两张表都有的键的并集进行联合

尽管不是很直观，但多对多的合并有明确的行为。下面是一个例子：

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                     'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                     'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b

In [48]: df2
Out[48]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
4      4   d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key  data2
0       0   b    1.0
1       0   b    3.0
2       1   b    1.0
3       1   b    3.0
4       2   a    0.0
5       2   a    2.0
```

```
6         3    c     NaN
7         4    a     0.0
8         4    a     2.0
9         5    b     1.0
10        5    b     3.0
```

多对多连接是行的笛卡尔积。由于在左边的DataFrame中有三个'b'行，而在右边有两行，因此在结果中有六个'b'行。连接方法仅影响结果中显示的不同键值：

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1 key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

使用多个键进行行合并时，传入一个列名的列表：

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                      'key2': ['one', 'two', 'one'],
   ....:                      'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                       'key2': ['one', 'one', 'one', 'two'],
   ....:                       'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[53]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

决定哪些键联合出现在结果中，取决于合并方法的选择，把多个键看作一个元组数据来作为单个连接键使用（尽管实际上并不是以这种方法来实现的）。

当你在进行列-列连接时，传递的DataFrame索引对象会被丢弃。

合并操作中最后一个要考虑的问题是如何处理重叠的列名。虽然你可以手动解决重叠问题（参见之前章节中的重命名轴标签的内容），但是merge有一个suffixes后缀选项，用于在左右两边DataFrame对象的重叠列名后指定需要添加的字符串：

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar    one     3    one     6
5  bar    one     3    two     7

In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right
Out[55]:
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
1  foo       one     1        one     5
2  foo       two     2        one     4
3  foo       two     2        one     5
4  bar       one     3        one     6
5  bar       one     3        two     7
```

表8-2是merge方法的参数参考。使用DataFrame的行索引进行连接是下一节的内容。

表8-2：merge函数参数

参数	描述
left	合并时操作中左边的 DataFrame
right	合并时操作中右边的 DataFrame
how	'inner'、'outer'、'left'、'right' 之一；默认是 'inner'
on	需要连接的列名。必须是在两边的 DataFrame 对象都有的列名，并以 left 和 right 中的列名的交集作为连接键
left_on	left DataFrame 中用作连接键的列
right_on	right DataFrame 中用作连接键的列
left_index	使用 left 的行索引作为它的连接键（如果是 MultiIndex，则是多个键）
right_index	使用 right 的行索引作为它的连接键（如果是 MultiIndex，则是多个键）
sort	通过连接键按字母顺序对合并的数据进行排序；在默认情况下为 True（在大数据集上某些情况下禁用该功能可以获得更好的性能）
suffixes	在重叠情况下，添加到列名后的字符串元组；默认是（'_x', '_y')（例如如果待合并的 DataFrame 中都含有 'data' 列，那么结果中会出现 'data_x'、'data_y'）
copy	如果为 False，则在某些特殊情况下避免将数据复制到结果数据结构中；默认情况下总是复制
indicator	添加一个特殊的列 _merge，指示每一行的来源；值将根据每行中连接数据的来源分别为 'left_only'，'right_only' 或 'both'

8.2.2　根据索引合并

在某些情况下，DataFrame中用于合并的键是它的索引。在这种情况下，你可以传递left_index＝True或right_index＝True（或者都传）来表示索引需要用来作为合并的键：

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                       'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a',

In [58]: left1
Out[58]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0

In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

由于默认的合并方法是连接键相交，你可以使用外连接来进行合并：

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how
Out[61]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

在多层索引数据的情况下，事情会更复杂，在索引上连接是一个隐式的多键合并：

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....:                                'Nevada', 'Nevada'],
   ....:                       'key2': [2000, 2001, 2002, 2001, 2002
   ....:                       'data': np.arange(5.)})

In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....:                      index=[['Nevada', 'Nevada', 'Ohio', 'Oh
   ....:                              'Ohio', 'Ohio'],
   ....:                             [2001, 2000, 2000, 2000, 2001,
                                       2002]],
   ....:                       columns=['event1', 'event2'])

In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002

In [65]: righth
Out[65]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
     2001       8       9
     2002      10      11
```

这种情况下，你必须以列表的方式指明合并所需多个列（请注意使用how='outer'处理重复的索引值）：

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_inde
Out[66]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000       4       5
0   0.0    Ohio  2000       6       7
1   1.0    Ohio  2001       8       9
2   2.0    Ohio  2002      10      11
3   3.0  Nevada  2001       0       1

In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:          right_index=True, how='outer')
```

```
Out[67]:
   data     key1  key2  event1  event2
0   0.0     Ohio  2000     4.0     5.0
0   0.0     Ohio  2000     6.0     7.0
1   1.0     Ohio  2001     8.0     9.0
2   2.0     Ohio  2002    10.0    11.0
3   3.0   Nevada  2001     0.0     1.0
4   4.0   Nevada  2002     NaN     NaN
4   NaN   Nevada  2000     2.0     3.0
```

使用两边的索引进行合并也是可以的：

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=['a', 'c', 'e'],
   ....:                      columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13
   ....:                       index=['b', 'c', 'd', 'e'],
   ....:                       columns=['Missouri', 'Alabama'])

In [70]: left2
Out[70]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [71]: right2
Out[71]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0

In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_
Out[72]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

DataFrame有一个方便的join实例方法，用于按照索引合并。该方法也可以用于合并多个索引相同或相似但没有重叠列的DataFrame对象。在之前的例子中，我们可以这样写：

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

由于一些历史原因（例如，一些非常早期版本的pandas），DataFrame的join方法进行连接键上的左连接，完全保留左边DataFrame的行索引。它还支持在调用DataFrame的某一列上连接传递的DataFrame的索引：

```
In [74]: left1.join(right1, on='key')
Out[74]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

最后，对于一些简单索引-索引合并，你可以向join方法传入一个DataFrame列表，这个方法可以替代下一节中将要介绍的使用更为通用的concat函数的方法：

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [1
   ....:                         index=['a', 'c', 'e', 'f'],
   ....:                         columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0

In [77]: left2.join([right2, another])
Out[77]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0
```

```
In [78]: left2.join([right2, another], how='outer')
Out[78]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
b   NaN     NaN       7.0      8.0       NaN     NaN
c   3.0     4.0       9.0     10.0       9.0    10.0
d   NaN     NaN      11.0     12.0       NaN     NaN
e   5.0     6.0      13.0     14.0      11.0    12.0
f   NaN     NaN       NaN      NaN      16.0    17.0
```

8.2.3　沿轴向连接

另一种数据组合操作可互换地称为拼接、绑定或堆叠。NumPy的concatenate函数可以在NumPy数组上实现该功能：

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

在Series和DataFrame等pandas对象的上下文中，使用标记的轴可以进一步泛化数组连接。尤其是你还有许多需要考虑的事情：

·如果对象在其他轴上的索引不同，我们是否应该将不同的元素组合在这些轴上，还是只使用共享的值（交集）？

·连接的数据块是否需要在结果对象中被识别？

·“连接轴”是否包含需要保存的数据？在许多情况下，DataFrame中的默认整数标签在连接期间最好丢弃。

pandas的concat函数提供了一种一致性的方式来解决以上问题。我将给出一些例子来表明它的工作机制。假设我们有三个索引不存在重叠的Series：

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

用列表中的这些对象调用concat方法会将值和索引粘在一起：

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

默认情况下，concat方法是沿着axis=0的轴向生效的，生成另一个Series。如果你传递axis=1，返回的结果则是一个DataFrame（axis=1时是列）：

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

在这个案例中另一个轴向上并没有重叠，你可以看到排序后的索引合集（'outer'join外连接）。你也可以传入join='inner'：

```
In [87]: s4 = pd.concat([s1, s3])
In [88]: s4
Out[88]:
a    0
b    1
f    5
g    6
dtype: int64

In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6

In [90]: pd.concat([s1, s4], axis=1, join='inner')
```

```
Out[90]:
   0  1
a  0  0
b  1  1
```

在这个例子中，由于join='inner'的选项，'f'和'g'标签消失了。

你甚至可以使用join_axes来指定用于连接其他轴向的轴：

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']
Out[91]:
     0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

拼接在一起的各部分无法在结果中区分是一个潜在的问题。假设你想在连接轴向上创建一个多层索引，可以使用keys参数来实现：

```
In [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three

In [93]: result
Out[93]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64

In [94]: result.unstack()
Out[94]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

沿着轴向axis=1连接Series的时候，keys则成为DataFrame的列头：

```
In [95]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'
Out[95]:
   one  two  three
a  0.0  NaN    NaN
```

```
b   1.0   NaN    NaN
c   NaN   2.0    NaN
d   NaN   3.0    NaN
e   NaN   4.0    NaN
f   NaN   NaN    5.0
g   NaN   NaN    6.0
```

将相同的逻辑拓展到DataFrame对象：

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a',
   ....:                         columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a
   ....:                         columns=['three', 'four'])

In [98]: df1
Out[98]:
   one  two
a    0    1
b    2    3
c    4    5

In [99]: df2
Out[99]:
   three  four
a      5     6
c      7     8

In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

如果你传递的是对象的字典而不是列表的话，则字典的键会用于keys选项：

```
In [101]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[101]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

还有一些额外的参数负责多层索引生成（见表8-3）。例如，我们可以使用names参数命名生成的轴层级：

```
In [102]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   .....:           names=['upper', 'lower'])
Out[102]:
upper level1     level2
lower    one two  three four
a          0   1    5.0  6.0
b          2   3    NaN  NaN
c          4   5    7.0  8.0
```

最后需要考虑的是行索引中不包含任何相关数据的DataFrame：

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

在这个示例中，我们传入ignore_index=True：


```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
          b         d         a
```

```
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

在这个示例中，我们传入ignore_index=True：

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

表8-3：concat 函数的参数

参数	描述
objs	需要连接的 pandas 对象列表或字典；这是必选参数
axis	连接的轴向；默认是 0（沿着行方向）
join	可以是 'inner' 或 'outer'（默认是 'outer'）；用于指定连接方式是内连接（inner）还是外连接（outer）
join_axes	用于指定其他 $n-1$ 轴的特定索引，可以替代内 / 外连接的逻辑
keys	与要连接的对象关联的值，沿着连接轴形成分层索引；可以是任意值的列表或数组，也可以是元组的数组，也可以是数组的列表（如果向 levels 参数传入多层数组）
leels	在键值传递时，该参数用于指定多层索引的层级

表8-3：concat 函数的参数（续）

参数	描述
names	如果传入了 keys 和 / 或 levels 参数，该参数用于多层索引的层级名称
verify_integrity	检查连接对象中的新轴是否重复，如果是，则引发异常；默认（False）允许重复
ignore_index	不沿着连接轴保留索引，而产生一段新的（长度为 total_length）索引

8.2.4 联合重叠数据

还有另一个数据联合场景，既不是合并操作，也不是连接操作。你可能有两个数据集，这两个数据集的索引全部或部分重叠。作为一个示例，考虑NumPy的where函数，这个函数可以进行面向数组的if-else等价操作：Series有一个combine_first方法，该方法可以等价于下面这种使用pandas常见数据对齐逻辑的轴向操作：

```
In [108]: a = pd,Serles([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
   .....:               index=['f', 'e', 'd', 'c', 'b', 'a'])

In [109]: b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
   .....:               index=['a', 'b', 'c', 'd', 'e', 'f'])

In [110]: a
Out[110]:
f    NaN
e    2.5
d    0.0
c    3.5
b    4.5
a    NaN
dtype: float64

In [111]: b
Out[111]:
a    0.0
b    NaN
c    2.0
d    NaN
e    NaN
f    5.0
dtype: float64

In [112]: np.where(pd.isnull(a), b, a)
Out[112]: array([ 0.  , 2.5, 0.  , 3.5, 4.5, 5.])
```

Series有一个combine_first方法，该方法可以等价于下面这种使用pandas常见数据对齐逻辑的轴向操作：

```
In [113]: b.combine_first(a)
Out[113]:
a 0.0
b 4.5
c 2.0
d 0.0
```

```
e 2.5
f 5.0
dtype: float64
```

在DataFrame中，combine_first逐列做相同的操作，因此你可以认为它是根据你传入的对象来”修补“调用对象的缺失值：

```
In [114]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
   .....:                     'b': [np.nan, 2., np.nan, 6.],
   .....:                     'c': range(2, 18, 4)})

In [115]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
   .....:                     'b': [np.nan, 3., 4., 6., 8.]})

In [116]: df1
Out[116]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [117]: df2
Out[117]:
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0

In [118]: df1.combine_first(df2)
Out[118]:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

8.3 重塑和透视

重排列表格型数据有多种基础操作。这些操作被称为重塑或透视。

8.3.1　使用多层索引进行重塑

多层索引在DataFrame中提供了一种一致性方式用于重排列数据。以下是两个基础操作：

statck（堆叠）

该操作会“旋转”或将列中的数据透视到行。

unstack（拆堆）

该操作会将行中的数据透视到列。

我将通过一系列的例子来说明这些操作。考虑一个带有字符串数组作为行和列索引的小型DataFrame：

```
In [119]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                  index=pd.Index(['Ohio', 'Colorado'], name=
   .....:                 columns=pd.Index(['one', 'two', 'three'],
   .....:                 name='number'))

In [120]: data
Out[120]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

在这份数据上使用stack方法会将列透视到行，产生一个新的Series：

```
In [121]: result = data.stack()

In [122]: result
Out[122]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

从一个多层索引序列中，你可以使用unstack方法将数据重排列后放入一个DataFrame中：

```
In [123]: result.unstack()
Out[123]:
number     one  two  three
state
Ohio         0    1      2
Colorado     3    4      5
```

默认情况下，最内层是已拆堆的（与stack方法一样）。你可以传入一个层级序号或名称来拆分一个不同的层级：

```
In [124]: result.unstack(0)
Out[124]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [125]: result.unstack('state')
Out[125]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

如果层级中的所有值并未包含于每个子分组中时，拆分可能会引入缺失值：

```
In [126]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [127]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [128]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [129]: data2
Out[129]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
```

```
       e    6
dtype: int64

In [130]: data2.unstack()
Out[130]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

默认情况下，堆叠会过滤出缺失值，因此堆叠拆堆的操作是可逆的：

```
In [131]: data2.unstack()
Out[131]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0

In [132]: data2.unstack().stack()
Out[132]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64

In [133]: data2.unstack().stack(dropna=False)
Out[133]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

当你在DataFrame中拆堆时，被拆堆的层级会变为结果中最低的层级：

```
In [134]: df = pd.DataFrame({'left': result, 'right': result + 5},
   .....:                   columns=pd.Index(['left', 'right'], name='s

In [135]: df
```

```
Out[135]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10

In [136]: df.unstack('state')
Out[136]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

在调用stack方法时，我们可以指明需要堆叠的轴向名称：

```
In [137]: df.unstack('state').stack('side')
Out[137]:
state         Colorado  Ohio
number side
one    left          3     0
       right         8     5
two    left          4     1
       right         9     6
three  left          5     2
       right        10     7
```

8.3.2　将“长”透视为“宽”

在数据库和CSV中存储多时间序列的方式就是所谓的长格式或堆叠格式。让我们载入一些示例数据，然后做少量的时间序列规整和其他的数据清洗操作：

```
In [138]: data = pd.read_csv('examples/macrodata.csv')

In [139]: data.head()
Out[139]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19

In [140]: periods = pd.PeriodIndex(year=data.year, quarter=data.quart
   .....:                          name='date')

In [141]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='iter

In [142]: data = data.reindex(columns=columns)

In [143]: data.index = periods.to_timestamp('D', 'end')

In [144]: ldata = data.stack().reset_index().rename(columns={0: 'valu
```

我们将在第11章中更深入地讲解PeriodIndex。简单地说，PeriodIndex将year和quarter等列进行联合并生成了一种时间间隔类型：

现在，ldata看起来如下：

```
In [145]: ldata[:10]
Out[145]:
        date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
```

```
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

这种数据即所谓的多时间序列的长格式，或称为具有两个或更多个键的其他观测数据（这里，我们的键是date和item）。表中的每一行表示一个时间点上的单个观测值。

数据通常以这种方式存储在关系型数据库中，比如MySQL，因为固定模式（列名称和数据类型）允许item列中不同值的数量随着数据被添加到表中而改变。在之前的例子中，date和item通常是主键（使用关系型数据库的说法），提供了关系完整性和更简单的连接。在某些情况下，处理这种格式的数据更为困难，你可能更倾向于获取一个按date列时间戳索引的且每个不同的item独立一列的DataFrame。DataFrame的pivot方法就是进行这种转换的：

```
In [146]: pivoted = ldata.pivot('date', 'item', 'value')

In [147]: pivoted
Out[147]:
item         infl    realgdp  unemp
date
1959-03-31   0.00   2710.349    5.8
1959-06-30   2.34   2778.801    5.1
1959-09-30   2.74   2775.488    5.3
1959-12-31   0.27   2785.204    5.6
1960-03-31   2.31   2847.699    5.2
1960-06-30   0.14   2834.390    5.2
1960-09-30   2.70   2839.022    5.6
1960-12-31   1.21   2802.616    6.3
1961-03-31  -0.40   2819.264    6.8
1961-06-30   1.47   2872.005    7.0
...           ...        ...    ...
2007-06-30   2.75  13203.977    4.5
2007-09-30   3.45  13321.109    4.7
2007-12-31   6.38  13391.249    4.8
2008-03-31   2.82  13366.865    4.9
2008-06-30   8.53  13415.266    5.4
2008-09-30  -3.16  13324.600    6.0
2008-12-31  -8.79  13141.920    6.9
2009-03-31   0.94  12925.410    8.1
2009-06-30   3.37  12901.504    9.2
2009-09-30   3.56  12990.341    9.6
[203 rows x 3 columns]
```

传递的前两个值是分别用作行和列索引的列，然后是可选的数值列以填充DataFrame。假设你有两个数值列，你想同时进行重塑：

```
In [148]: ldata['value2'] = np.random.randn(len(ldata))

In [149]: ldata[:10]
Out[149]:
        date     item     value    value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31     infl     0.000  0.000940
2 1959-03-31    unemp     5.800  1.343810
3 1959-06-30  realgdp  2778.801 -0.713544
4 1959-06-30     infl     2.340 -0.831154
5 1959-06-30    unemp     5.100 -2.370232
6 1959-09-30  realgdp  2775.488 -1.860761
7 1959-09-30     infl     2.740 -0.860757
8 1959-09-30    unemp     5.300  0.560145
9 1959-12-31  realgdp  2785.204 -1.265934
```

如果遗漏最后一个参数，你会得到一个含有多层列的DataFrame：

```
In [150]: pivoted = ldata.pivot('date', 'item')

In [151]: pivoted[:5]
Out[151]:
           value                    value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543

In [152]: pivoted['value'][:5]
Out[152]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

请注意，pivot方法等价于使用set_index创建分层索引，然后调用unstack：

```
In [153]: unstacked = ldata.set_index(['date', 'item']).unstack('iten

In [154]: unstacked[:7]
Out[154]:
           value                      value2
item        infl   realgdp unemp        infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543
1960-06-30  0.14  2834.390   5.2 -0.970736 -1.541996 -1.307030
1960-09-30  2.70  2839.022   5.6  0.377984  0.286350 -0.753887
```

8.3.3 将“宽”透视为“长”

在DataFrame中，pivot方法的反操作是pandas.melt。与将一列变换为新的Data Frame中的多列不同，它将多列合并成一列，产生一个新的DataFrame，其长度比输入更长。让我们看下面这个例子：

```
In [156]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})

In [157]: df
Out[157]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

'key'列可以作为分组指标，其他列均为数据值。当使用pandas.melt时，我们必须指明哪些列是分组指标（如果有的话）。此处，让我们使用'key'作为唯一的分组指标：

```
In [158]: melted = pd.melt(df, ['key'])

In [159]: melted
Out[159]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

使用pivot方法，我们可以将数据重塑回原先的布局：

```
In [160]: reshaped = melted.pivot('key', 'variable', 'value')

In [161]: reshaped
Out[161]:
```

```
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

由于pivot的结果根据作为行标签的列生成了索引，我们可能会想要使用reset_index来将数据回移一列：

```
In [162]: reshaped.reset_index()
Out[162]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

你也可以指定列的子集作为值列：

```
In [163]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[163]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

pandas.melt的使用也可以无须任何分组指标：

```
In [164]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[164]:
  variable  value
0        A      1
1        A      2
2        A      3
3        B      4
4        B      5
5        B      6
6        C      7
7        C      8
8        C      9

In [165]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[165]:
  variable value
```

```
0      key   foo
1      key   bar
2      key   baz
3        A     1
4        A     2
5        A     3
6        B     4
7        B     5
8        B     6
```

8.4　本章小结

目前你已经拥有了pandas的基础知识，包括数据导入、清洗以及按照你自己的想法重新组织数据，我们已经预备好去使用matplotlib进行数据可视化了。在后续章节中讨论高阶分析时，我们将会重返pandas的内容。

第9章　绘图与可视化

制作提供信息的可视化（有时称为绘图）是数据分析中最重要任务之一。可视化可能是探索过程的一部分，例如，帮助识别异常值或所需的数据转换，或者为建模提供一些想法。对于其他人来说，构建网络交互式可视化可能是最终目标。Python有很多附加库可以用来制作静态或动态的可视化文件，但是我将主要关注matplotlib（http://matplotlib.sourceforge.net ）和以它为基础的库。

matplotlib是一个用于生成出版级质量图表（通常是二维的）的桌面绘图包。该项目由John Hunter于2002年发起，目的在于在Python环境下进行MATLAB风格的绘图。matplotlib和IPython社区合作简化了IPython shell（目前是Jupyter notebook）的交互式绘图。matplotlib支持所有操作系统上的各种GUI后端，还可以将可视化导出为所有常见的矢量和光栅图形格式（PDF、SVG、JPG、PNG、BMP、GIF等）。除了少数图表之外，本书中几乎所有的图形都是使用matplotlib生成的。

随着时间的推移，matplotlib已经产生了一些数据可视化的附加工具包，使用matplotlib进行底层绘图。seaborn（http://seaborn.pydata.org ）就是其中之一，我们将在本章后面介绍。

学习本章示例代码最简单的方式就是在Jupyter notebook中使用交互式绘图。在进行设置时，需要在Jupyter notebook中执行以下语句：

```
%matplotlib notebook
```

9.1 简明matplotlib API入门

在使用matplotlib时，我们使用以下的导入惯例：

```
In [11]: import matplotlib.pyplot as plt
```

在Jupyter中运行%matplotlib notebook（或在IPython中运行%matplotlib），我们就可以尝试生成一个简单的图形。如果所有的设置都正确，则一个像图9-1的图形就会出现：

```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

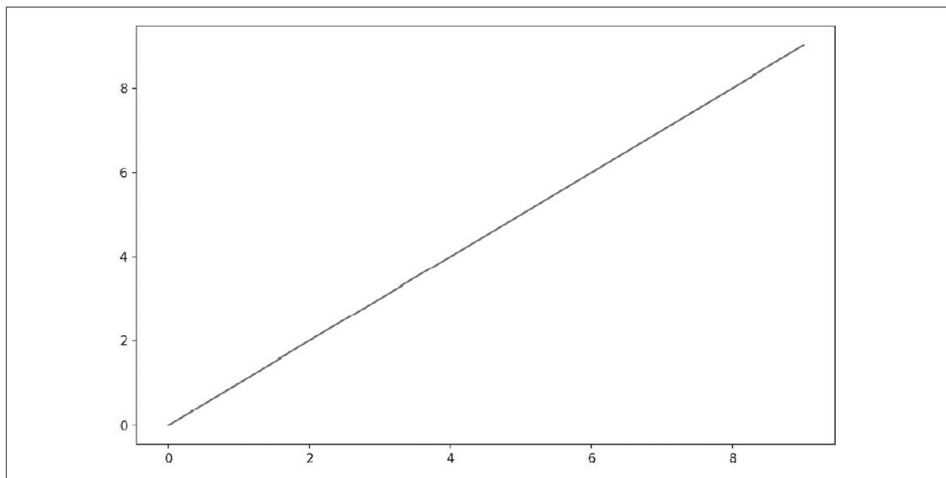


图9-1：简单的线性图

尽管seaborn等库和pandas内建的绘图函数可以处理大部分绘图的普通细节，但如果你想在提供的函数选项之外进行定制则需要学习一些matplotlib的API。

 本书没有足够的篇幅来对matplotlib的功能广度和深度进行全面介绍。但介绍的内容应该是足以使你入门的。matplotlib的可视化作品库和文档是学习高级功能的最佳资源。

9.1.1 图片与子图

matplotlib所绘制的图位于图片（Figure）对象中。你可以使用plt.figure生成一个新的图片：

```
In [16]: fig = plt.figure()
```

在IPython中，一个空白的绘图窗口就会出现，但在Jupyter中则没有任何显示，直到我们使用一些其他命令。plt.figure有一些选项，比如figsize是确保图片有一个确定的大小以及存储到硬盘时的长宽比。

你不能使用空白的图片进行绘图。你需要使用add_subplot创建一个或多个子图（subplot）：

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

上面代码的意思是图片应该是2×2的（最多四个图形），并且我们选择了四个图形中的第一个（序号从1开始）。如果你接着创建了两个子图，你将会获得看上去类似图9-2的可视化：

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)

In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

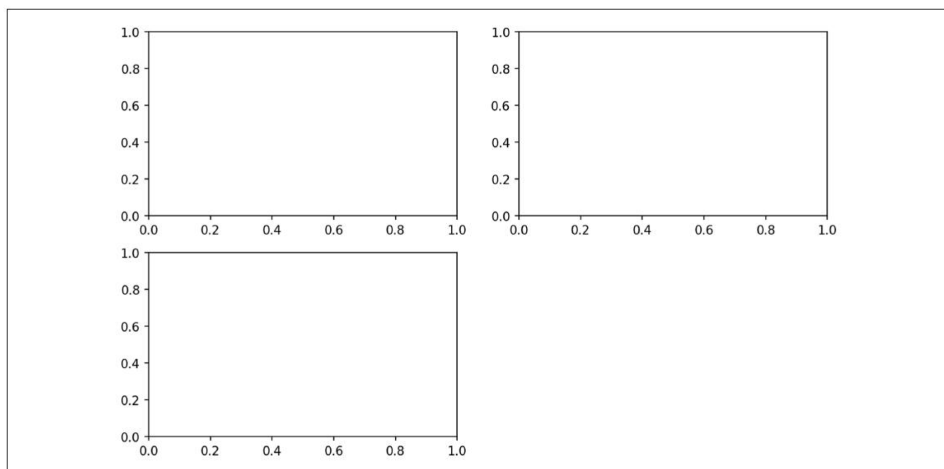


图9-2：一个带有三个子图的空白matplotlib图片

使用Jupyter notebook时有个细节需要注意，在每个单元格运行后，图表被重置，因此对于更复杂的图表，你必须将所有的绘图命令放在单个的notebook单元格中。

我们将下面这些代码在同一个单元格中运行：

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

当你输入绘图命令plt.plot（[1.5，3.5，-2，1.6]），matplotlib会在最后一个图片和子图（如果需要的话就创建一个）上进行绘制，从而隐藏图片和子图的创建。因此，如果我们添加了下面的代码，你会得到形如图9-3的可视化：

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

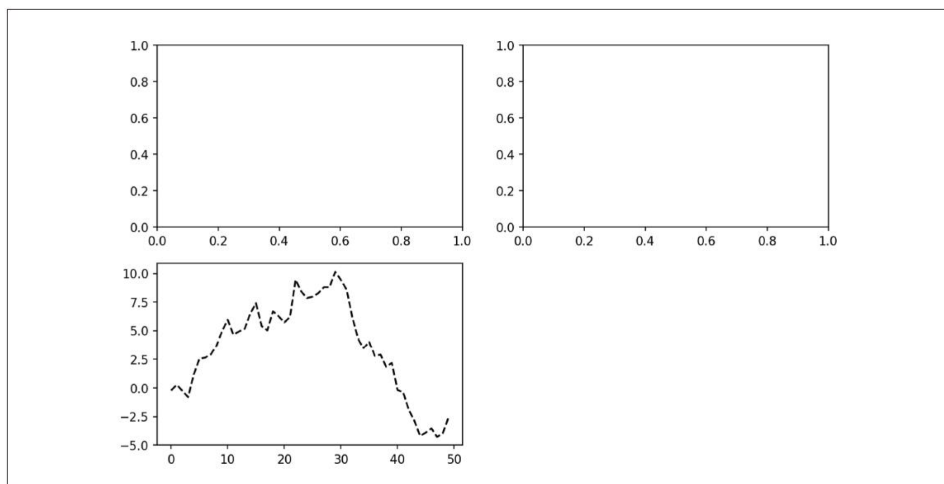


图9-3：单个子图绘制的数据可视化

'k--'是用于绘制黑色分段线的style选项。fig.add_subplot返回的对象是Axes Subplot对象，使用这些对象你可以直接在其他空白的子图上调用对象的实例方法进行绘图（参考图9-4）：

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha

In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.ra
```

你可以在matplotlib的官方文档（http://matplotlib.sourceforge.net）中找到完整的图形类型。

使用子图网格创建图片是非常常见的任务，所以matplotlib包含了一个便捷方法plt.subplots，它创建一个新的图片，然后返回包含了已生成子图对象的NumPy数组：

```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625d
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a3
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd
```

```
=object)
```

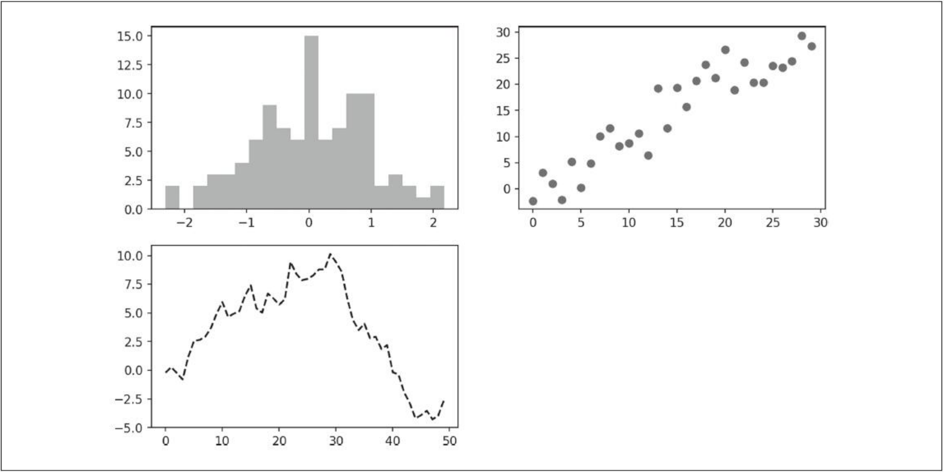


图9-4：增加子图后的数据可视化

这是非常有用的，因为数组axes可以像二维数组那样方便地进行索引，例如，axes[0，1]。你也可以通过使用sharex和sharey来表明子图分别拥有相同的x轴或y轴。当你在相同的比例下进行数据对比时，sharex和sharey会非常有用；此外，matplotlib还可以独立地缩放图像的界限。表9-1对此方法有更多介绍。

表9-1：pyplot.subplots选项

参数	描述
nrows	子图的行数
ncols	子图的列数
sharex	所有子图使用相同的 x 轴刻度（调整 xlim 会影响所有子图）
sharey	所有子图使用相同的 y 轴刻度（调整 ylim 会影响所有子图）
subplot_kw	传入 add_subplot 的关键字参数字典，用于生成子图
**fig_kw	在生成图片时使用的额外关键字参数，例如 plt.subplots（2, 2, figsize=（8,6））

9.1.1.1　调整子图周围的间距

默认情况下，matplotlib会在子图的外部和子图之间留出一定的间距。这个间距都是相对于图的高度和宽度来指定的，所以如果你通过编程

或手动使用GUI窗口来调整图的大小，那么图就会自动调整。你可以使用图对象上的subplots_adjust方法更改间距，也可以用作顶层函数：

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

wspace和hspace分别控制的是图片的宽度和高度百分比，以用作子图间的间距。下面是一个小例子，我将这个间距一直缩小到零（见图9-5）：

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50, color='k', alp
plt.subplots_adjust(wspace=0, hspace=0)
```

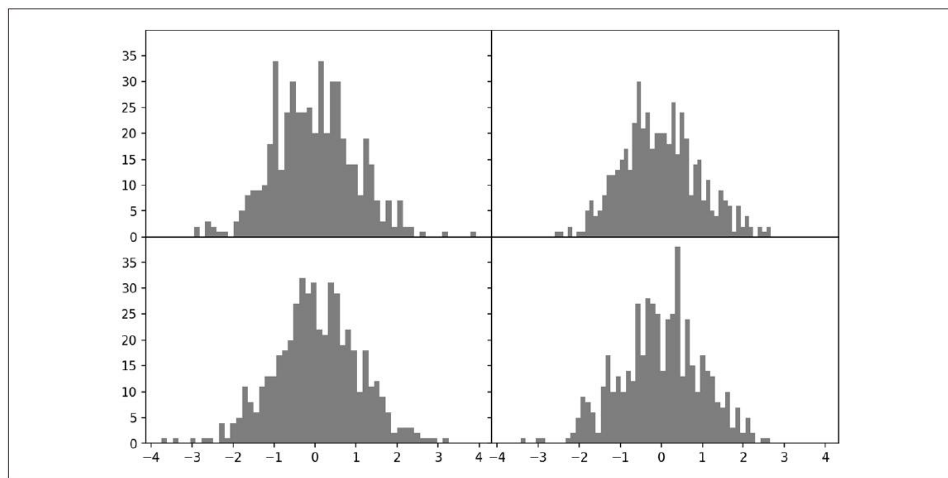


图9-5：没有内部子图间隔的数据可视化

你可能会注意到轴标签是存在重叠的。matplotlib并不检查标签是否重叠，因此在类似情况下你需要通过显式指定刻度位置和刻度标签的方法来修复轴标签（我们将在下一节中讲解如何实现该操作）。

9.1.2　颜色、标记和线类型

matplotlib的主函数plot接收带有x和y轴的数组以及一些可选的字符串缩写参数来指明颜色和线类型。例如，要用绿色破折号绘制x对y的线，你需要执行：

```
ax.plot(x, y, 'g--')
```

这种在字符串中指定颜色和线条样式的方式是方便的。在实践中，如果你以编程方式创建绘图，则可能不希望将字符串混合在一起以创建具有所需样式的图表。同样的图表可以使用更为显式的方式来表达：

```
ax.plot(x, y, linestyle='--', color='g')
```

有很多颜色缩写被用于常用颜色，但是你可以通过指定十六进制颜色代码的方式来指定任何颜色（例如'#CECECE'）。参考plot函数的文档字符串可以看到全部的线类型（在IPython或Jupyter中使用plot？）。

折线图还可以有标记用来凸显实际的的数据点。由于matplotlib创建了一个连续性折线图，插入点之间有时并不清除点在哪。标记可以是样式字符串的一部分，样式字符串中线类型、标记类型必须跟在颜色后面（参考图9-6）：

```
In [30]: from numpy.random import randn

In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

图9-6：带有标记的折线图

上面的代码可以写得更为显式：

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

对于折线图，你会注意到后续的点默认是线性内插的。可以通过drawstyle选项进行更改（图9-7）：

```
In [33]: data = np.random.randn(30).cumsum()
In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]

In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-po

Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]
In [36]: plt.legend(loc='best')
```

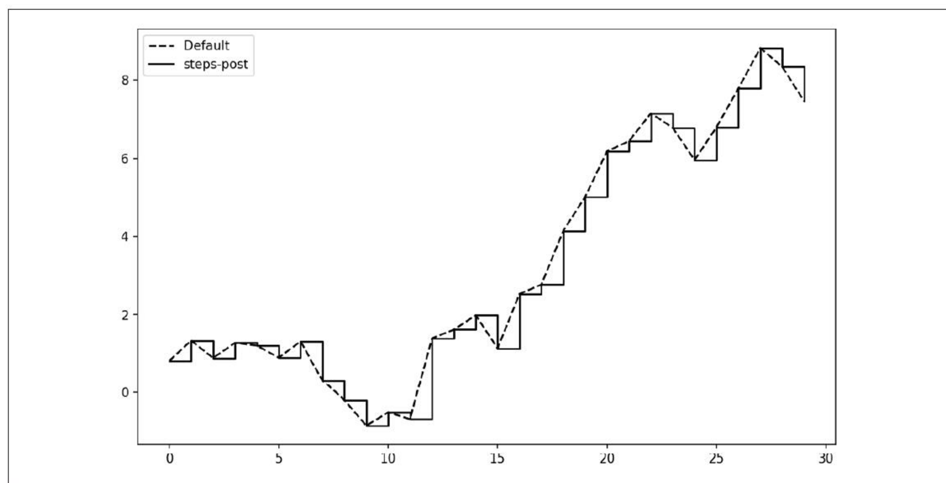


图9-7：不同drawstyle选项下的折线图

你可能会注意到在运行代码后会有像＜matplotlib.lines.Line2D at...＞这样的输出。matplotlib返回的对象引用了刚刚添加的图表子组件。很多时候你可以安全地忽略这些输出。这里，由于我们向plot传递了label，我们可以使用plt.legend为每条线生成一个用于区分的图例。

无论你在用数据绘图时是否传递了lebel选项，你都必须调用plt.legend（如果你有轴的引用，也可以用ax.legend）来生成图例。

9.1.3 刻度、标签和图例

对于大多数图表修饰工作，有两种主要的方式：使用程序性的pyplot接口（即matplotlib.pyplot）和更多面向对象的原生matplotlib API。

pyplot接口设计为交互式使用，包含了像xlim、xticks和xticklabels等方法。这些方法分别控制了绘图范围、刻度位置以及刻度标签。我们可以在两种方式中使用：

·在没有函数参数的情况下调用，返回当前的参数值（例如plt.xlim（）返回当前的x轴绘图范围）。

·传入参数的情况下调用，并设置参数值（例如plt.xlim（[0，10]）会将x轴的范围设置为0到10）。

所有的这些方法都会在当前活动的或最近创建的AxesSubplot上生效。这些方法中的每一个对应于子图自身的两个方法。比如xlim对应于ax.get_lim和ax.set_lim。我更倾向于使用subplot的实例方法，因为这样更为显式（尤其是在处理多个子图时），但你当然可以使用你觉得更为方便的方式。

9.1.3.1 设置标题、轴标签、刻度和刻度标签

为了讲解轴的自定义，我会生成一个简单图表，并绘制随机漫步（参考图9-8）：

```
In [37]: fig = plt.figure()

In [38]: ax = fig.add_subplot(1, 1, 1)

In [39]: ax.plot(np.random.randn(1000).cumsum())
```

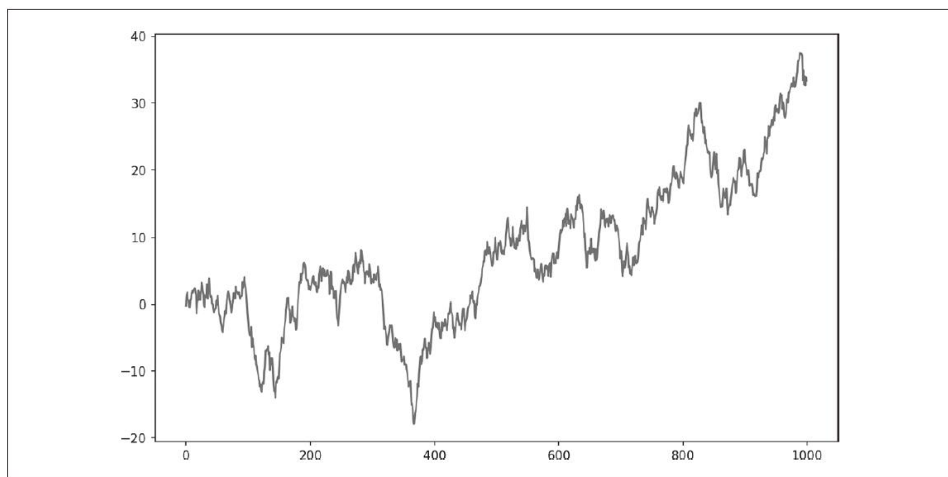


图9-8：表述x轴（以及轴标签）的简单图表

要改变x轴刻度，最简单的方式是使用set_xticks和set_xticklabels。set_xticks表示在数据范围内设定刻度的位置，默认情况下，这些刻度也有标签。但是我们可以使用set_xticklabels为标签赋值：

```
In [40]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
In [41]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
   ....:                             rotation=30, fontsize='small')
```

rotation选项会将x轴刻度标签旋转30度。最后，set_xlabel会给x轴一个名称，set_titel会给子图一个标题（参考图9-9的结果图）：

```
In [42]: ax.set_title('My first matplotlib plot')
Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>

In [43]: ax.set_xlabel('Stages')
```

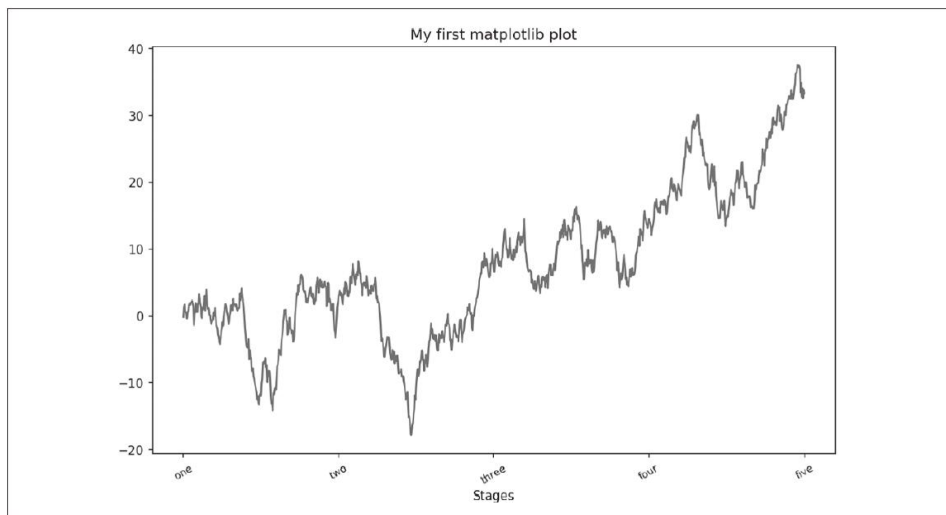


图9-9：x轴刻度的简单示例

修改y轴坐标是相同过程，将上面示例中的x替换成y即可。轴的类型拥有一个set方法，允许批量设置绘图属性。根据之前的示例，我们可以写如下代码：

```
props = {
    'title': 'My first matplotlib plot',
    'xlabel': 'Stages'
}
ax.set(**props)
```

9.1.3.2 添加图例

图例是用来区分绘图元素的另一个重要内容。有多种方式可以添加图例。最简单的方式是在添加每个图表时传递label参数：

```
In [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
```

```
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]
```

一旦你运行了上面的代码，你也可以调用ax.legend（）或plt.legend自动生成图例。结果图表参见图9-10。

```
In [49]: ax.legend(loc='best')
```

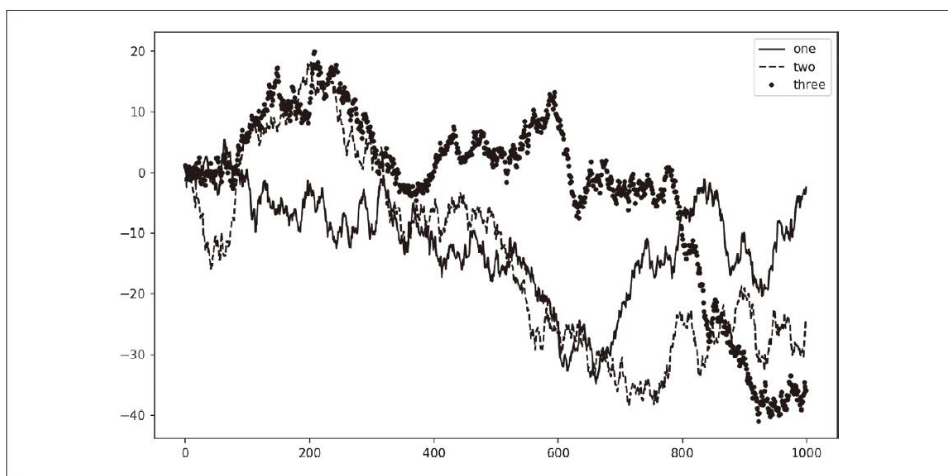


图9-10：有三根折线和图例的简单图表

legend方法有多个其他的位置参数loc。参考文档字符串（使用ax.legend？命令）获取更多信息。

loc参数告诉matplotlib在哪里放置图表。如果你不挑剔，'best'是一个好选项，它会自动选择最合适的位置。如果取消图例中的元素，不要传入label参数或者传入label='_nolegend_'。

9.1.4 注释与子图加工

除了标准的绘图类型，你可能还会想在图表上绘制自己的注释，而且注释中可能会包含文本、箭头以及其他图形。你可以使用text、arrow和annote方法来添加注释和文本。text在图表上给定的坐标（x，y），根据可选的定制样式绘制文本：

```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

注释可以同时绘制文本和箭头。作为一个例子，让我们绘制标普500指数从2007年以来的收盘价（从雅虎财经获得数据），并在图表中标注从2008年到2009年金融危机中的重要日期。你可以在Jupyter notebook的一个单元格中复现这些代码。参考图9-11的代码运行结果：

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', headwidth=4, widt
                                headlength=4),
                horizontalalignment='left', verticalalignment='top')

# 放大2007年~2010年
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in the 2008-2009 financial crisis')
```

在图表中有一些重要点需要凸显：ax.annotate方法可以在指定的x和y坐标上绘制标签。我们可以使用set_xlim和set_ylim方法手动设置图表的边界，而不是使用matplotlib的默认设置。最后，ax.set_title给图表添加了一个主标题。

参考在线的matplotlib展览馆，可以学习更多注释的范例。

绘制图形时有更多需要注意的地方。matplotlib含有表示多种常见图形的对象，这些对象的引用是patches。一些图形，比如Rectangle（矩形）和Circle（圆形），可以在matplotlib.pyplot中找到，但图形的全集位于matplotlib.patches。

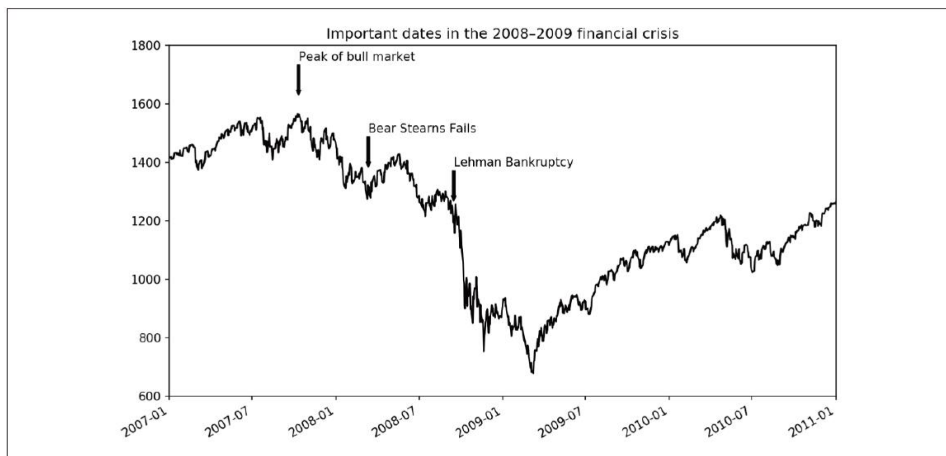


图9-11：2008——2009金融危机中的重要日期

想在图表中添加图形时，你需要生成patch（补丁）对象shp，并调用ax.add_patch（shp）将它加入到子图中（参考图9-12）：

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

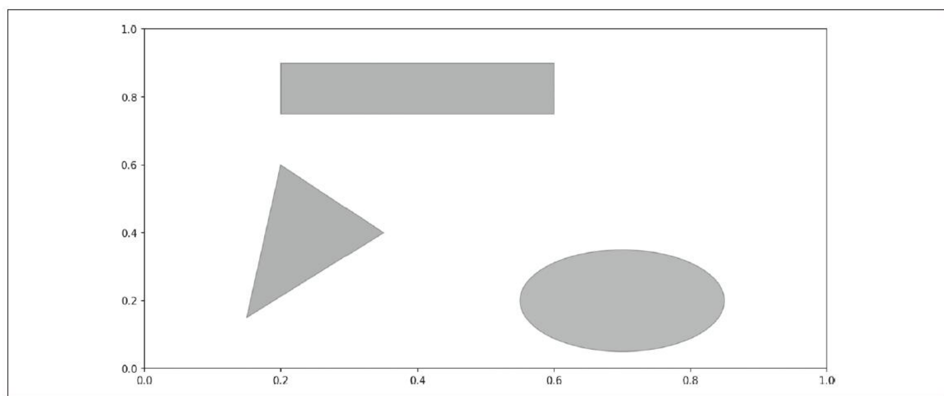


图9-12：三个不同patch图形的可视化

当你看到很多常见绘图类型的实现时，你会发现他们都是由patches组装而来。

9.1.5　将图片保存到文件

你可以使用plt.savefig将活动图片保存到文件。这个方法等价于图片对象的savefig实例方法。例如将图片保存为SVG，你只需要输入以下代码：

```
plt.savefig('figpath.svg')
```

文件类型是从文件扩展名中推断出来的。所以如果你使用.pdf，则会得到一个PDF。我常常使用几个重要的选项来发布图形：dpi，它控制每英寸点数的分辨率；bbox_inches，可以修剪实际图形的空白。为了得到同样一个PNG图片，且使用最小的空白，拥有400DPI，你需要运行以下代码：

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

saveifg并非一定是写到硬盘的，它可以将图片写入到所有的文件型对象中，例如BytesIO：

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

表9-2是savefig其他选项的列表。

表9-2：figure.savefig选项

参数	描述
fname	包含文件路径或 Python 文件型对象的字符串。图片格式是从文件扩展名中推断出来的（例如 PDF 格式的 .pdf 或 PNG 的 .png 格式）
dpi	每英寸点数的分辨率；默认为 100，但可以配置
facecolor,edgecolor	子图之外的图形背景的颜色；默认情况下是 'w'（白色）
format	文件格式（'png', 'pdf', 'svg', 'ps', 'eps'......）
bbox_inches	要保存的图片范围；如果传递 'tight'，将会去除掉图片周围空白的部分

9.1.6 matplotlib设置

matplotlib配置了配色方案和默认设置，主要用来准备用于发布的图片。幸运的是，几乎所有的默认行为都可以通过广泛的全局参数来定制，包括图形大小、子图间距、颜色、字体大小和网格样式等等。使用rc方法是使用Python编程修改配置的一种方式，例如，要将全局默认数字大小设置为10×10，你可以输入：

```
plt.rc('figure', figsize=(10, 10))
```

rc的第一个参数是你想要自定义的组件，比如'figure'、'axes'、'xtick'、'ytick'、'grid'、'legend'等等。之后，可以按照关键字参数的序列指定新参数。字典是一种在程序中设置选项的简单方式：

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

如果需要更深入的定制和参看全量选项，可以参考matplotlib的设置文件matplotlibrc，该文件位于matplotlib/mpl-data路径。如果你定制了这个文件，并将他放置在home路径下并且文件名为.matplotlibrc，则每次你使用matplotlib时都会读取该文件。

下一节中我们会看到，seaborn包中的若干内建绘图主题或样式，seaborn内部使用了matplotlib的设置系统。

9.2 使用pandas和seaborn绘图

matplotlib是一个相当底层的工具。你可以从其基本组件中组装一个图表：数据显示（即绘图的类型：线、条、框、散点图、轮廓等）、图例、标题、刻度标记和其他注释。

在pandas中，我们可能有多个数据列，并且带有行和列的标签。pandas自身有很多内建方法可以简化从DataFrame和Series对象生成可视化的过程。另一个库是seaborn（https://seaborn.pydata.org/ ），它是由Michael Waskom创建的统计图形库。seaborn简化了很多常用可视化类型的生成。

导入seaborn会修改默认的matplotlib配色方案和绘图样式，这会提高图表的可读性和美观性。即使你不使用seaborn的API，你可能更喜欢导入seaborn来为通用matplotlib图表提供更好的视觉美观度。

9.2.1　折线图

Series和DataFrame都有一个plot属性，用于绘制基本的图形。默认情况下，plot（）绘制的是折线图（见图9-13）：

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange
In [61]: s.plot()
```

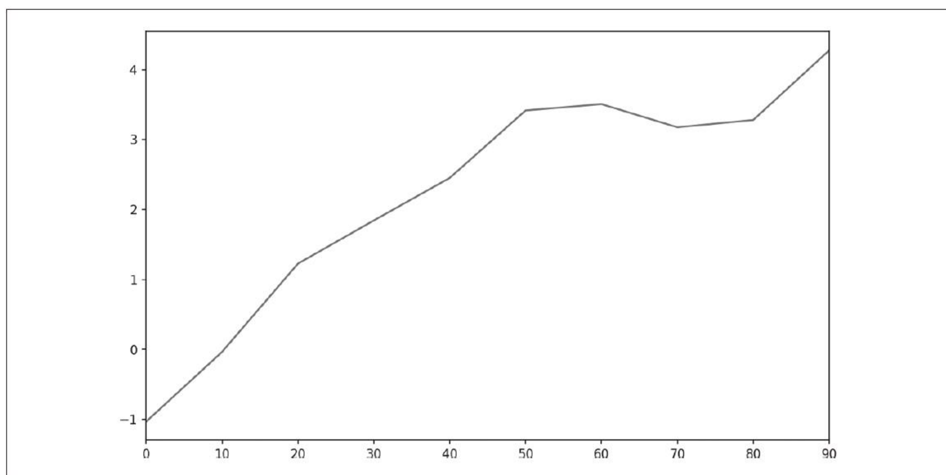


图9-13：简单序列图形

Series对象的索引传入matplotlib作为绘图的x轴，你可以通过传入use_index＝False来禁用这个功能。x轴的刻度和范围可以通过xticks和xlim选项进行调整，相应地y轴使用yticks和ylim进行调整。表9-3是plot的全部选项列表。本节我会介绍这些选项中的一些，其余你可以自行探索。

大部分pandas的绘图方法，接收可选的ax参数，该参数可以是一个matplotlib子图对象。这使你可以更为灵活地在网格布局中放置子图。

DataFrame的plot方法在同一个子图中将每一列绘制为不同的折线，并自动生成图例（见图9-14）：

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
   ....:                   columns=['A', 'B', 'C', 'D'],
   ....:                   index=np.arange(0, 100, 10))
```

```
In [63]: df.plot()
```

plot属性包含了不同绘图类型的方法族。例如，df.plot（ ）等价于df.plot.line（ ）。我们之后将会探索这些方法中的一部分。

plot的其他关键字参数会传递到相应的matplotlib绘图函数，因此你可以通过了解更多的matplotlib的API信息来进一步定制这些图表。

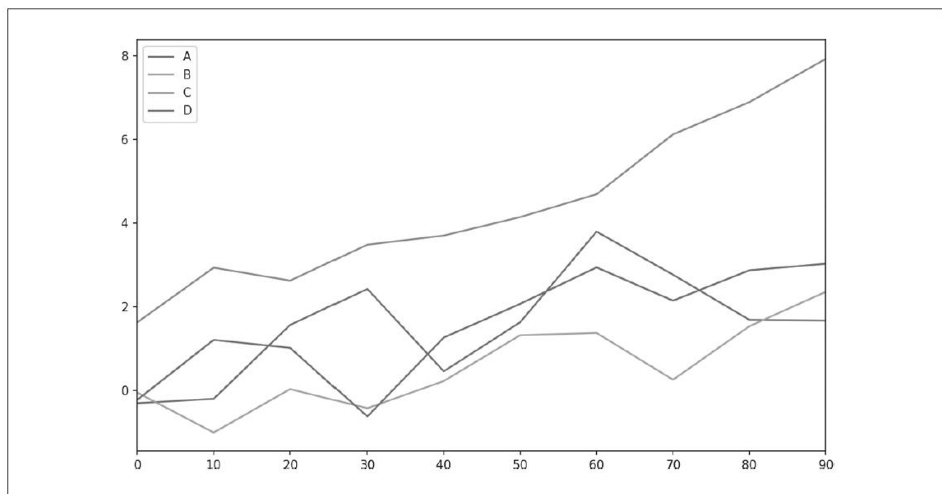


图9-14：简单DataFrame绘图

表9-3：Series.plot方法参数

参数	描述
label	图例标签
ax	绘图所用的 matplotlib 子图对象；如果没传值，则使用当前活动的 matplotlib 子图
style	传给 matplotlib 的样式字符串，比如 'ko--'
alpha	图片不透明度（从 0 到 1）
kind	可以是 'area'、'bar'、'barh'、'density'、'hist'、'kde'、'line'、'pie'
logy	在 y 轴上使用对数缩放
use_index	使用对象索引刻度标签
rot	刻度标签的旋转（0 到 360）
xticks	用于 x 轴刻度的值
yticks	用于 y 轴刻度的值
xlim	x 轴范围（例如 [0,10]）
ylim	y 轴范围
grid	展示轴网格（默认是打开的）

DataFrame拥有多个选项，允许灵活地处理列。例如，是否将各列绘制到同一个子图中，或为各列生成独立的子图。参考表9-4了解更多选项。

表9-4：DataFrame的plot参数

参数	描述
subplots	将 DataFrame 的每一列绘制在独立的子图中
sharex	如果 subplots=True，则共享相同的 x 轴、刻度和范围
sharey	如果 subplots=True，则共享相同的 y 轴
figsize	用于生成图片尺寸的元组
title	标题字符串
legend	添加子图图例（默认是 True）
sort_columns	按字母顺序绘制各列，默认情况下使用已有的列顺序

 时间序列绘图，将在第11章介绍。

9.2.2　柱状图

plot.bar（）和plot.barh（）可以分别绘制垂直和水平的柱状图。在绘制柱状图时，Series或DataFrame的索引将会被用作x轴刻度（bar）或y轴刻度（barh）（参考图9-15）：

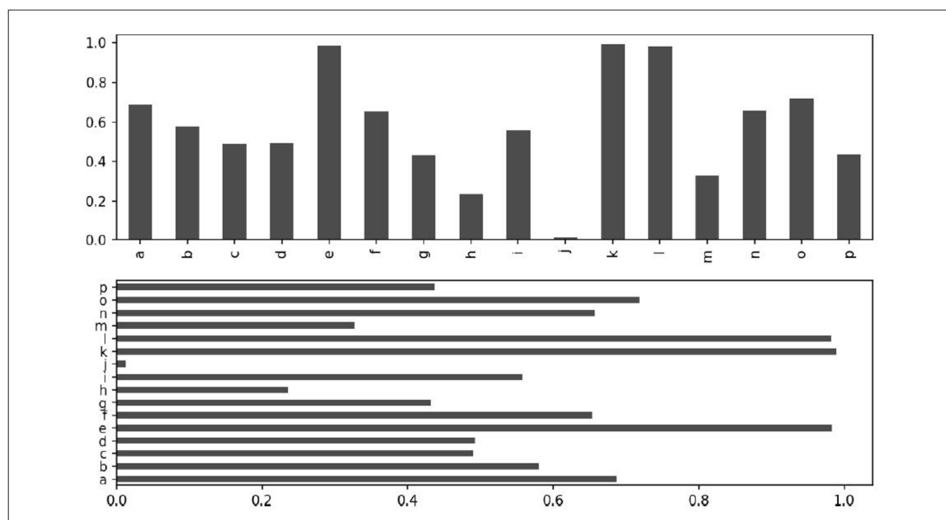


图9-15：水平柱状图和垂直柱状图

```
In [64]: fig, axes = plt.subplots(2, 1)

In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijl
In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb62493d470>

In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

选项color='k'和alpha=0.7将柱子的颜色设置为黑色，并将图像的填充色设置为部分透明。

在DataFrame中，柱状图将每一行中的值分组到并排的柱子中的一组。参考图9-16：

```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
   ....:                   index=['one', 'two', 'three', 'four', 'five
```

```
   ....:                    columns=pd.Index(['A', 'B', 'C', 'D'], nar

In [70]: df
Out[70]:
Genus         A         B         C         D
one    0.370670  0.602792  0.229159  0.486744
two    0.420082  0.571653  0.049024  0.880592
three  0.814568  0.277160  0.880316  0.431326
four   0.374020  0.899420  0.460304  0.100843
five   0.433270  0.125107  0.494675  0.961825
six    0.601648  0.478576  0.205690  0.560547

In [71]: df.plot.bar()
```

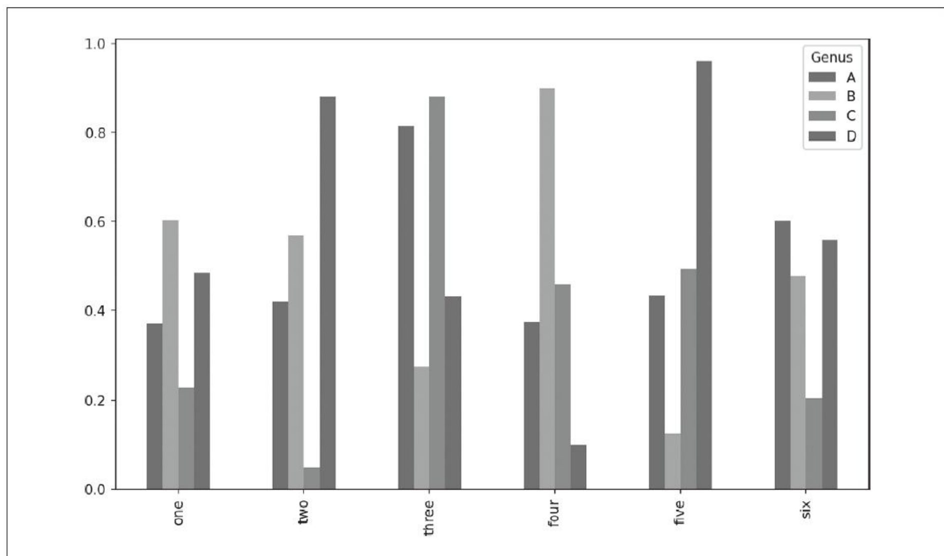


图9-16：DataFrame柱状图

请注意DataFrame的列名称"Genus"被用作了图例标题。我们可以通过传递stacked＝True来生成堆积柱状图，会使得每一行的值堆积在一起（参考图9-17）：

```
In [73]: df.plot.barh(stacked=True, alpha=0.5)
```

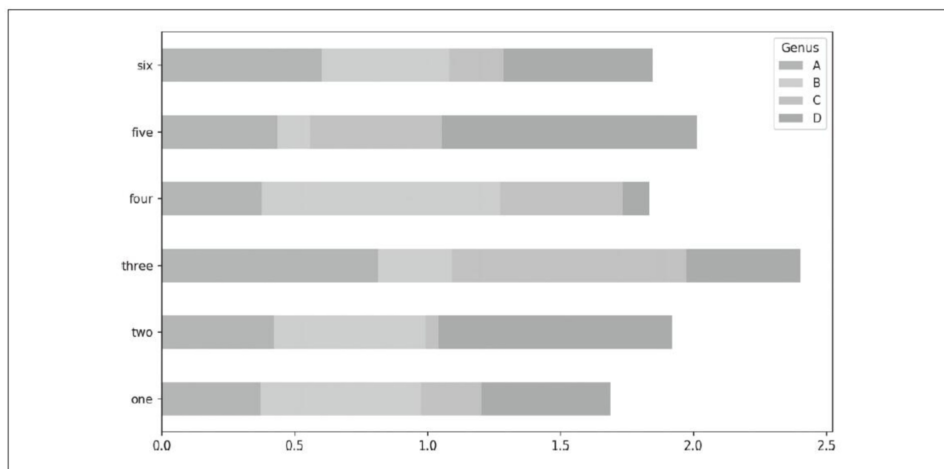


图9-17：DataFrame堆积柱状图

使用value_counts：s.value_counts（）.plot.bar（）可以有效地对Series值频率进行可视化。

回到本书之前使用的数据集，假设我们想要绘制一个堆积柱状图，用于展示每个派对在每天的数据点占比。使用read_csv载入数据，并根据星期日期和派对规模形成交叉表：

```
In [75]: tips = pd.read_csv('examples/tips.csv')

In [76]: party_counts = pd.crosstab(tips['day'], tips['size'])

In [77]: party_counts
Out[77]:
size  1   2   3   4  5  6
day
Fri   1  16   1   1  0  0
Sat   2  53  18  13  1  0
Sun   0  39  15  18  3  1
Thur  1  48   4   5  1  3

# 没有太多的1人和6人派对
In[78]: party_counts = party_counts.loc[:, 2:5]
```

之后，进行标准化以确保每一行的值和为1，然后进行绘图（见图9-18）：

```
# 标准化至和为1
In [79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)

In [80]: party_pcts
Out[80]:
size         2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241

In [81]: party_pcts.plot.bar()
```

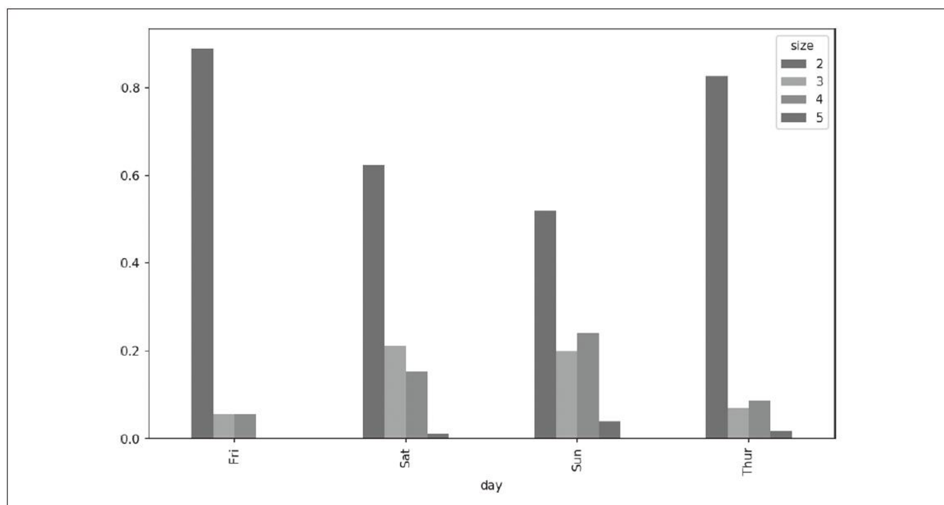


图9-18：每天分规模的派对数量百分比

你可以看到本数据集中的派对数量在周末会增加。

对于在绘图前需要聚合或汇总的数据，使用seaborn包会使工作更为简单。现在让我们看下使用seaborn进行按星期日期计算小费百分比（见图9-19中的结果图）：

```
In [83]: import seaborn as sns

In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips[

In [85]: tips.head()
Out[85]:
   total_bill   tip smoker  day    time  size   tip_pct
```

```
0        16.99  1.01     No  Sun  Dinner     2  0.063204
1        10.34  1.66     No  Sun  Dinner     3  0.191244
2        21.01  3.50     No  Sun  Dinner     3  0.199886
3        23.68  3.31     No  Sun  Dinner     2  0.162494
4        24.59  3.61     No  Sun  Dinner     4  0.172069

In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

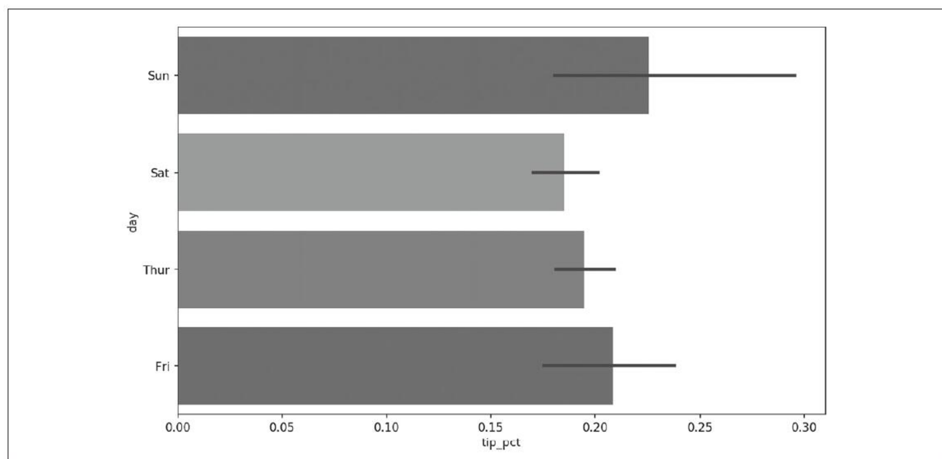


图9-19：用错误栏按天显示小费百分比

seaborn中的绘图函数使用一个data参数，这个参数可以是pandas的DataFrame。其他的参数则与列名有关。因为day列中有多个观测值，柱子的值是tip_pct的平均值。柱子上画出的黑线代表的是95%的置信区间（置信区间可以通过可选参数进行设置）。

seaborn.barplot拥有一个hue选项，允许我们通过一个额外的分类值将数据分离（见图9-20）：

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, or
```

请注意seaborn自动改变了图表的美观性：默认的调色板、图背景和网格线条颜色。你可以使用seaborn.set在不同的绘图外观中进行切换：

```
In [90]: sns.set(style="whitegrid")
```

9.2.3 直方图和密度图

直方图是一种条形图，用于给出值频率的离散显示。数据点被分成离散的，均匀间隔的箱，并且绘制每个箱中数据点的数量。使用之前的小费数据，我们可以使用Series的plot.hist方法制作小费占总费用百分比的直方图（见图9-21）：

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

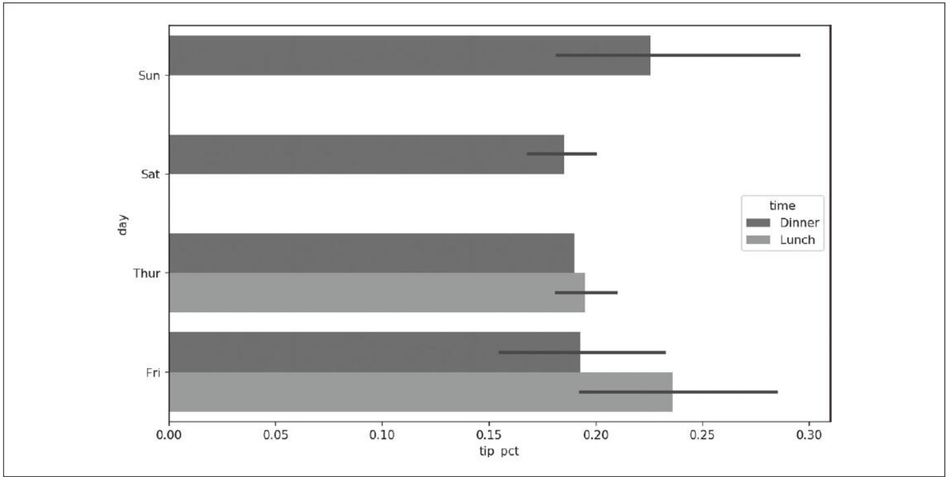


图9-20：根据星期日期和时间计算的小费百分比

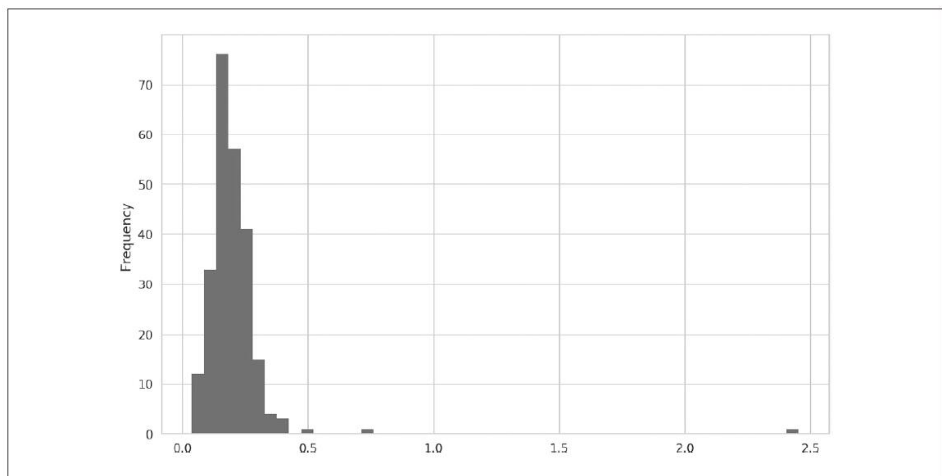


图9-21：小费百分比的直方图

密度图是一种与直方图相关的图表类型，它通过计算可能产生观测数据的连续概率分布估计而产生。通常的做法是将这种分布近似为“内核”的混合，也就是像正态分布那样简单的分布。因此，密度图也被称为内核密度估计图（KDE）。plot.kde使用传统法定混合法估计绘制密度图（见图9-22）：

```
In [94]: tips['tip_pct'].plot.density()
```

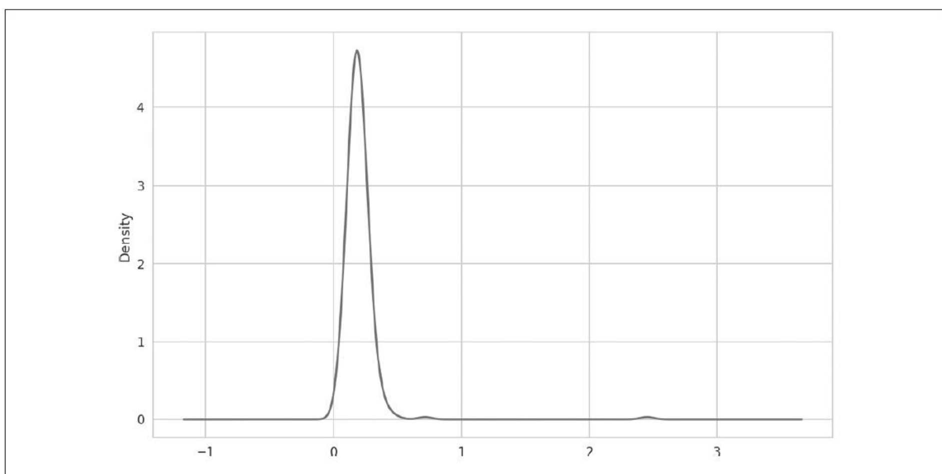


图9-22：小费百分比密度图

distplot方法可以绘制直方图和连续密度估计，通过distplot方法seaborn使直方图和密度图的绘制更为简单。作为例子，考虑由两个不同的标准正态分布组成的双峰分布（见图9-23）：

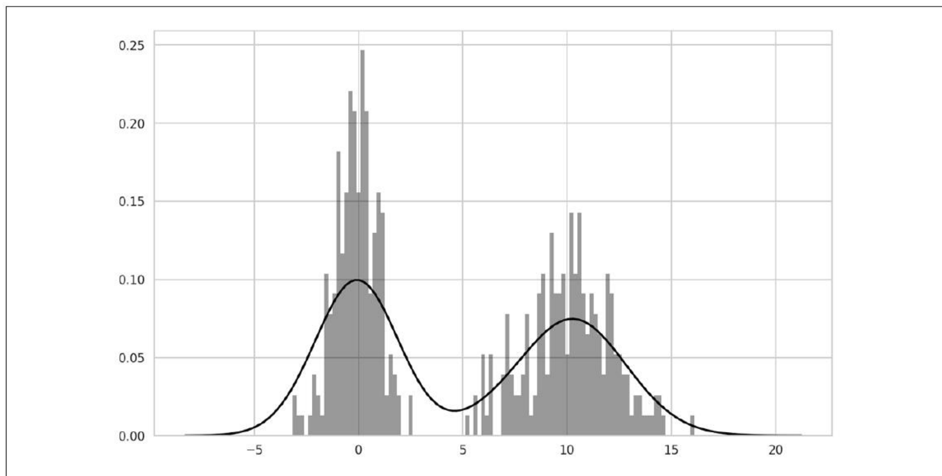


图9-23：正态混合的标准化直方图与密度估计

```
In [96]: comp1 = np.random.normal(0, 1, size=200)

In [97]: comp2 = np.random.normal(10, 2, size=200)

In [98]: values = pd.Series(np.concatenate([comp1, comp2]))

In [99]: sns.distplot(values, bins=100, color='k')
```

9.2.4 散点图或点图

点图或散点图可以用于检验两个一维数据序列之间的关系。例如，这里我们从statsmodels项目中载入了macrodata数据集，并选择了一些变量，之后计算对数差：

```
In [100]: macro = pd.read_csv('examples/macrodata.csv')

In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

In [102]: trans_data = np.log(data).diff().dropna()

In [103]: trans_data[-5:]
Out[103]:
          cpi        m1  tbilrate     unemp
198 -0.007904  0.045361 -0.396881  0.105361
199 -0.021979  0.066753 -2.277267  0.139762
200  0.002340  0.010286  0.606136  0.160343
201  0.008419  0.037461 -0.200671  0.127339
202  0.008894  0.012202 -0.405465  0.042560
```

然后我们可以使用seaborn的regplot方法，该方法可以绘制散点图，并拟合出一个条线性回归线（见图9-24）：

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb613720be0>

In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp
```

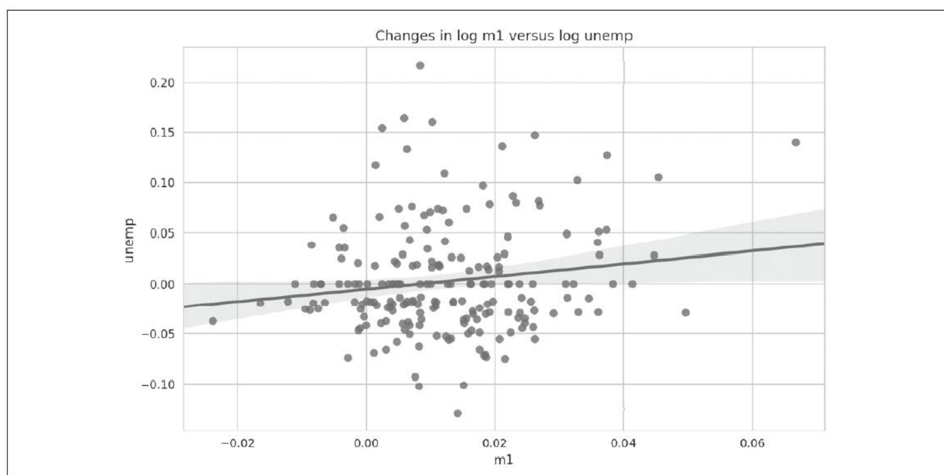


图9-24：seaborn回归/散点图

在探索性数据分析中，能够查看一组变量中的所有散点图是有帮助的，这被称为成对图或散点图矩阵。从头开始绘制这样一个图是有点工作量的，所以seaborn有一个方便的pairplot函数，它支持在对角线上放置每个变量的直方图或密度估计值（结果图见图9-25）：

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha
```

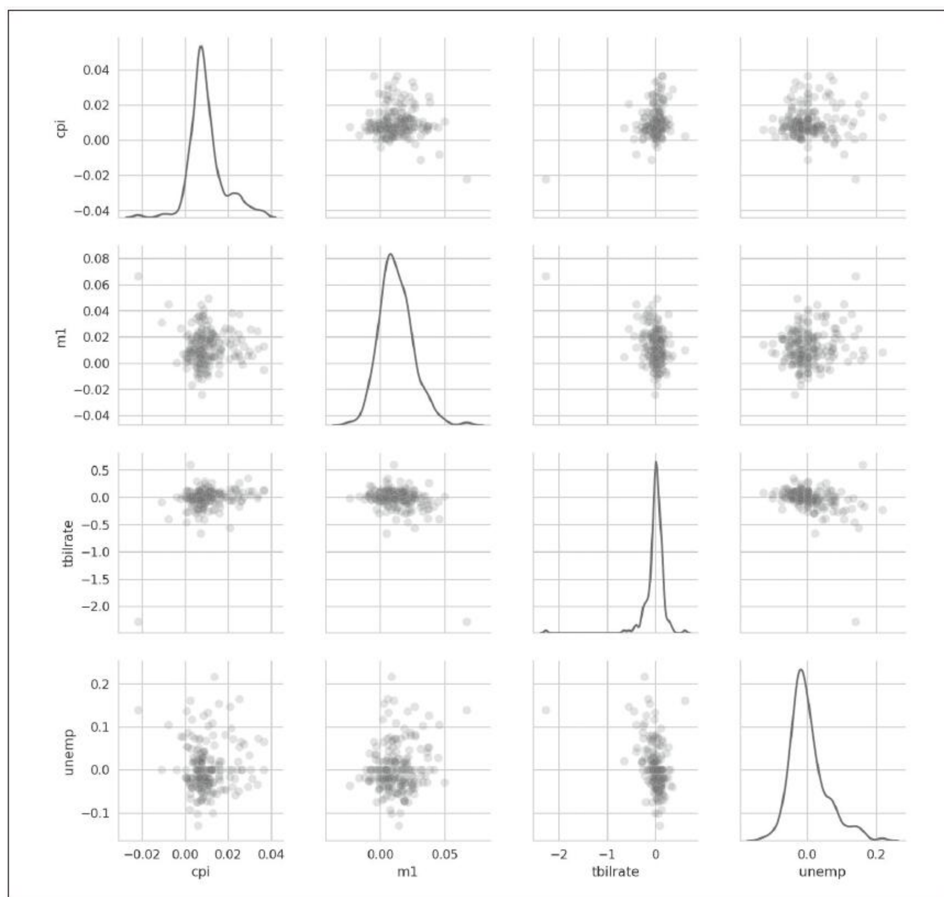


图9-25：statsmodels macro数据的成对图矩阵

你可能会注意到plot_ksw参数，这个参数使我们能够将配置选项传递给非对角元素上的各个绘图调用。参考seaborn.pairplot的文档字符串可以看到更多细节的设置选项

9.2.5 分面网格和分类数据

如果数据集有额外的分组维度怎么办？使用分面网格是利用多种分组变量对数据进行可视化的方式。seaborn拥有一个有效的内建函数factorplot，它可以简化多种分面绘图（见图9-26）：

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='time', col='smoke
   .....:                kind='bar', data=tips[tips.tip_pct < 1])
```

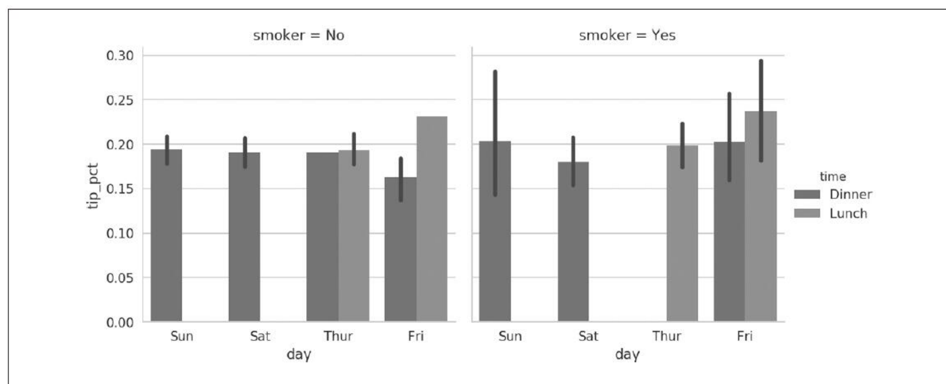


图9-26：按星期日期/时间/是否吸烟划分的小费百分比

除了根据'time'在一个面内将不同的柱分组为不同的颜色，我们还可以通过每个时间值添加一行来扩展分面网格（见图9-27）：

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='time',
   .....:                col='smoker',
   .....:                kind='bar', data=tips[tips.tip_pct < 1])
```

factorplot支持其他可能有用的图类型，具体取决于你要显示的内容。例如，箱形图（显示中位值、四分位数和异常值）可以是有效的可视化类型（图9-28）：

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='box',
   .....:                data=tips[tips.tip_pct < 0.5])
```

你可以使用更通用的seaborn.FacetGrid类创建自己的分面网格图。具体请查看更多的seaborn文档（https://seaborn.pydata.org/ ）。

9.3 其他Python可视化工具

和开源代码一样，在Python语言下创建图形的选择有很多（太多而无法一一列举）。自2010年以来，很多开发工作都集中在创建web交互式图形上。借助像Bokeh（http://bokeh.pydata.org/ ）和Plotly（https://github.com/plotly/plotly.py ）这样的工具，在web浏览器中创建动态的、交互式图像的工作现在已经可以实现。

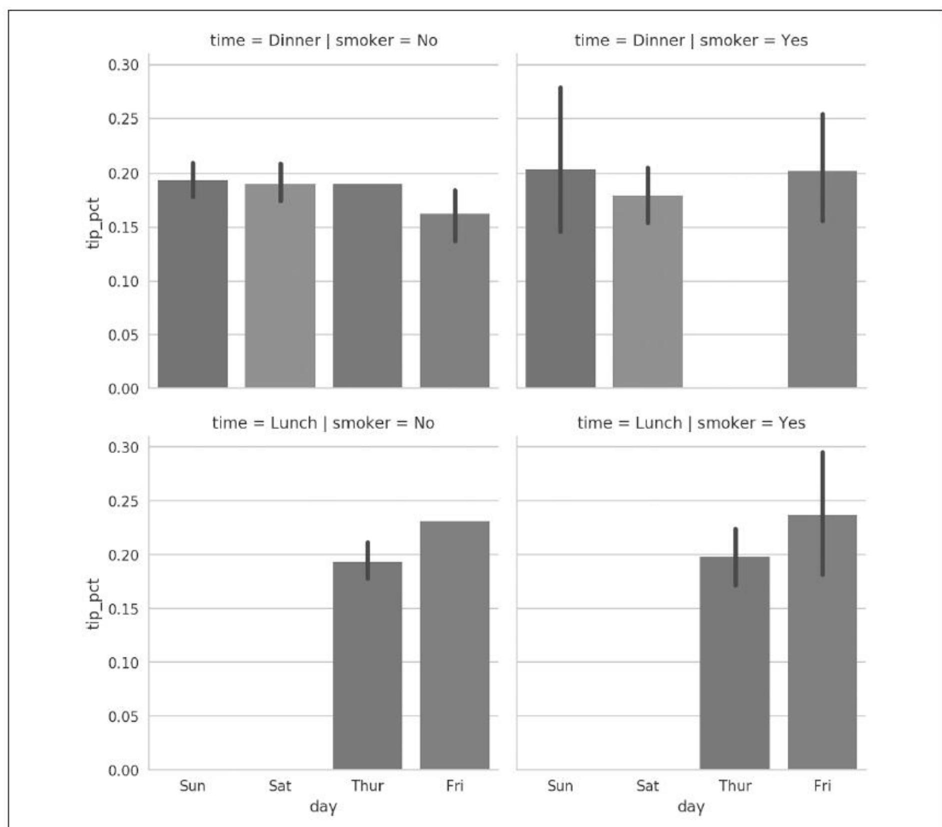


图9-27：根据时间/是否吸烟分面后按星期日期划分的小费百分比

如果是创建用于印刷或网页的静态图形，我建议根据你的需要使用默认的matplotlib以及像pandas和seaborn这样的附加库。对于其他数据可视化要求，学习其他可用工具之一可能是有用的。我鼓励你探索Python可视化生态系统，因为它将持续增添新内容并在未来进行更多创新。

9.4　本章小结

本章的目的是让你使用pandas、matplotlib和seaborn入门一些基础的数据可视化知识。如果直观地传达数据分析结果在你的工作中非常重要，我建议你查找资源以了解更多有关数据可视化的信息。可视化是一个活跃的研究领域，你可以实践在网上和书本上的许多优秀学习资源。

下一章，我将把注意力放在使用pandas进行数据聚合与分组操作上。

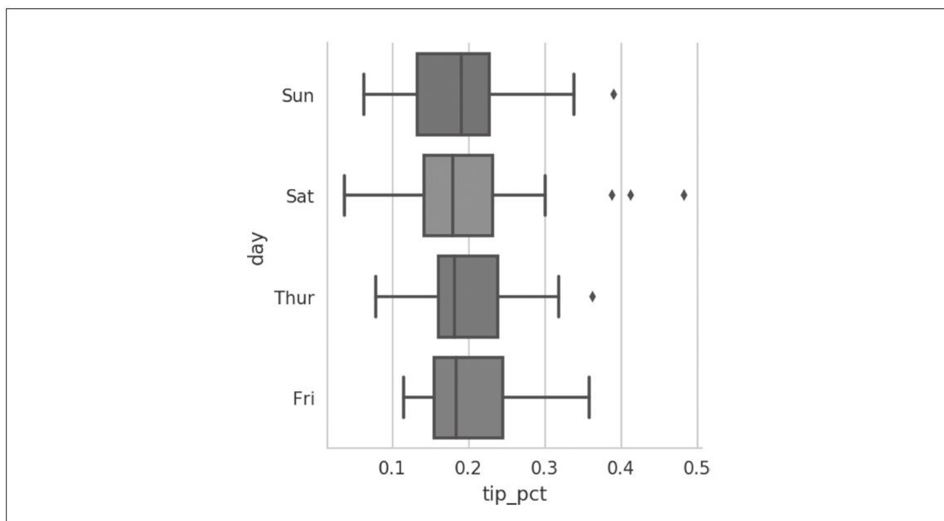


图9-28：根据星期日期绘制的小费百分比箱形图

第10章　数据聚合与分组操作

对数据集进行分类，并在每一组上应用一个聚合函数或转换函数，这通常是数据分析工作流中的一个重要部分。在载入、合并、准备数据集之后，你可能需要计算分组统计或者数据透视表用于报告或可视化的目的。pandas提供一个灵活的groupby接口，允许你以一种自然的方式对数据集进行切片、切块和总结。

关系型数据库和SQL（Structured Query Language，结构化查询语言）的流行原因之一就是其对数据的连接、过滤、变换和聚合功能。但是，像SQL这样的查询语言在可以执行的组操作种类上有所限制。正如你将看到的，通过Python和pandas的表达，我们可以使用pandas对象或NumPy数组执行相当复杂的组操作。在本章，你将学习如何：

·使用一个或多个键（以函数、数组或DataFrame列名的形式）将pandas对象拆分为多块

·计算组汇总统计信息，如计数、平均值或标准偏差或用户定义的函数

·应用组内变换或其他操作，如标准化、线性回归、排位或子集选择

·计算数据透视表和交叉表

·执行分位数分析和其他统计组分析

时间序列数据的聚合是groupby的特殊用例，它在本书中称为重采样，将在第11章中单独介绍。

10.1 GroupBy机制

Hadley Wickham是许多流行R语言软件包的作者，他创造了用于描述组操作的术语拆分-应用-联合（split-apply-combine）。在操作的第一步，数据包含在pandas对象中，可以是Series、DataFrame或其他数据结构，之后根据你提供的一个或多个键分离到各个组中。分离操作是在数据对象的特定轴向上进行的。例如，DataFrame可以在它的行方向（axis＝0）或列方向（axis＝1）进行分组。分组操作后，一个函数就可以应用到各个组中，产生新的值。最终，所有函数的应用结果会联合为一个结果对象。结果对象的形式通常取决于对数据进行的操作。图10-1是一个简单的分组聚合样例：

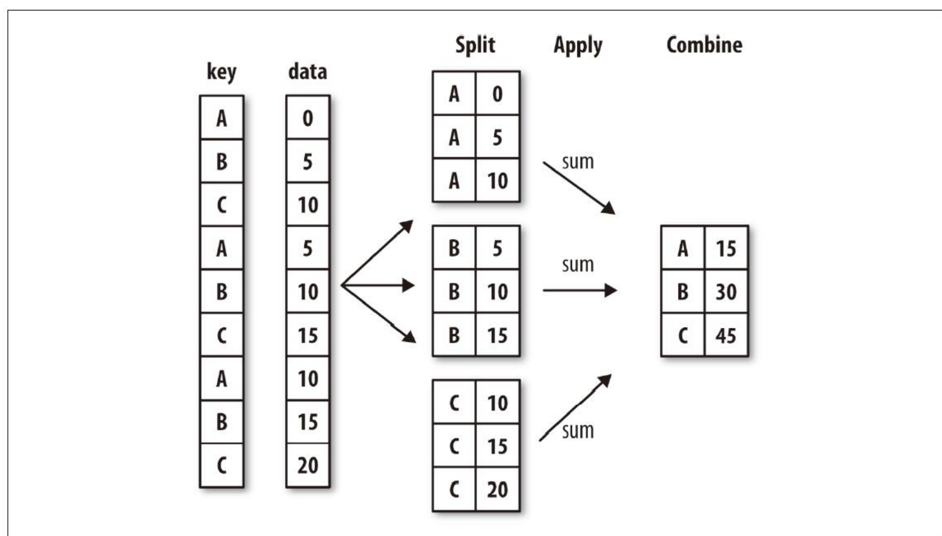


图10-1：分组聚合图示

分组键可是多种形式的，并且键不一定是完全相同的类型：

·与需要分组的轴向长度一致的值列表或值数组

·DataFrame的列名的值

·可以将分组轴向上的值和分组名称相匹配的字典或Series

·可以在轴索引或索引中的单个标签上调用的函数

请注意后面介绍的三个方法是可以产生用于分隔对象的值数组的快捷方式。如果感到抽象也不必担心。在本章中，我将给出这些方法的许多例子。下面是一个小型表格数据集作为DataFrame的例子：

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                    'key2' : ['one', 'two', 'one', 'two', 'o
   ....:                    'data1' : np.random.randn(5),
   ....:                    'data2' : np.random.randn(5)})

In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

假设你想要根据key1标签计算data1列的均值，有多种方法可以实现。其中一种是访问data1并使用key1列（它是一个Series）调用groupby方法：

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390
```

grouped变量现在是一个GroupBy对象。除了一些关于分组键df['key1']的一些中间数据之外，它实际上还没有进行任何计算。这个对象拥有所有必需的信息，之后可以在每一个分组上应用一些操作。例如，为了计算分组的均值我们可以调用GroupBy的mean方法：

```
In [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

之后，我们会更多地解释调用.mean（）时发生了什么。数据（一个Series）根据分组键进行了聚合，并产生了一个新的Series，这个Series使用key1列的唯一值作为索引。由于DataFrame的列df['key1']，结果中的索

引名称是'key1'。

如果我们将多个数组作为列表传入，则我们会得到一些不同的结果：

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean(

In [16]: means
Out[16]:
key1  key2
a     one     0.880536
      two     0.478943
b     one    -0.519439
      two    -0.555730
Name: data1, dtype: float64
```

这里我们使用了两个键对数据进行分组，并且结果Series现在拥有一个包含唯一键对的多层索引：

```
In [17]: means.unstack()
Out[17]:
key2       one       two
key1
a     0.880536  0.478943
b    -0.519439 -0.555730
```

在这个例子中，分组键都是Series，尽管分组键也可以是正确长度的任何数组：

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohi

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
            2006    1.965781
Name: data1, dtype: float64
```

分组信息作为你想要继续处理的数据，通常包含在同一个DataFrame中。在这种情况下，你可以传递列名（无论那些列名是字符串、数字或其他Python对象）作为分组键：

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
              data1     data2
key1 key2
a    one   0.880536  1.319920
     two   0.478943  0.092908
b    one  -0.519439  0.281746
     two  -0.555730  0.769023
```

你可能已经注意到第一行代码中df.groupby（'key1'）.mean（）的结果里并没有key2列。这是因为df['key2']并不是数值数据，即df['key2']是一个冗余列，因此被排除在结果之外。默认情况下，所有的数值列都可以聚合，尽管可能会过滤到子集，你将在后面的内容看到。

如果不在意使用groupby的目的，通用的GroupBy方法是size，size方法返回一个包含组大小信息的Series：

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

请注意，分组键中的任何缺失值将被排除在结果之外。

10.1.1 遍历各分组

GroupBy对象支持迭代，会生成一个包含组名和数据块的2维元组序列。考虑以下代码：

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

在多个分组键的情况下，元组中的第一个元素是键值的元组：

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

当然，你可以选择在任何一块数据上进行你想要的操作。使用一行代码计算出数据块的字典可能会对你有用：

```
In [26]: pieces = dict(list(df.groupby('key1')))
```

```
In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

默认情况下，groupby在axis＝0的轴向上分组，但你也可以在其他任意轴向上进行分组。例如，我们可以像以下代码一样，根据dtype对我们的示例df的列进行分组：

```
In [28]: df.dtypes
Out[28]:
data1    float64
data2    float64
key1      object
key2      object
dtype: object

In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

我们可以打印各分组，如下：

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
   ....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

10.1.2　选择一列或所有列的子集

将从DataFrame创建的GroupBy对象用列名称或列名称数组进行索引时，会产生用于聚合的列子集的效果。这表明：

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

是下面代码的语法糖：

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

尤其是对于大型数据集，可能只需要聚合少部分列。例如，在处理数据集时，要计算data2列的均值，并获得DataFrame形式的结果，我们可以写：

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

如果传递的是列表或数组，则此索引操作返回的对象是分组的DataFrame；如果只有单个列名作为标量传递，则为分组的Series：

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa30c78da0>

In [34]: s_grouped.mean()
Out[34]:
key1  key2
a     one     1.319920
      two     0.092908
b     one     0.281746
```

```
      two     0.769023
Name: data2, dtype: float64
```

10.1.3　使用字典和Series分组

分组信息可能会以非数组形式存在。让我们考虑另一个示例DataFrame：

```
DataFrame：
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....:                       columns=['a', 'b', 'c', 'd', 'e'],
   ....:                       index=['Joe', 'Steve', 'Wes', 'Jim',

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

现在，假设我拥有各列的分组对应关系，并且想把各列按组累加：

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

现在，你可以根据这个字典构造传给groupby的数组，但是我们也可以直接传字典（我多写了键'f'用于表明未用的分组键也是没问题的）：

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
            blue       red
Joe     0.503905  1.063885
Steve   1.297183 -1.553778
Wes    -1.021228 -1.116829
Jim     0.524712  1.770545
Travis -4.230992 -2.405455
```

Series也有相同的功能，可以视为固定大小的映射：

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a       red
b       red
c      blue
d      blue
e       red
f    orange
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

10.1.4 使用函数分组

与使用字典或Series分组相比，使用Python函数是定义分组关系的一种更为通用的方式。作为分组键传递的函数将会按照每个索引值调用一次，同时返回值会被用作分组名称。更具体地说，考虑上一节中的示例DataFrame，其中人的名字作为索引值。假设你想根据名字的长度来进行分组。虽然你可以计算出字符串长度的数组，但传递len函数更为简单：

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

将函数与数组、字典或Series进行混合并不困难，所有的对象都会在内部转换为数组：

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

10.1.5 根据索引层级分组

分层索引的数据集有一个非常方便的地方，就是能够在轴索引的某个层级上进行聚合。让我们看一个例子：

```
In [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP
   ....:                                      [1, 3, 5, 1, 3]],
   ....:                                      names=['cty', 'tenor'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=colum

In [49]: hier_df
Out[49]:
cty          US                            JP
tenor         1         3         5         1         3
0      0.560145 -1.265934  0.119827 -1.063512  0.332883
1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2      0.286350  0.377984 -0.753887  0.331286  1.349742
3      0.069877  0.246674 -0.011862  1.004812  1.327195
```

根据层级分组时，将层级数值或层级名称传递给level关键字：

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

10.2　数据聚合

聚合是指所有根据数组产生标量值的数据转换过程。之前的例子已经使用了一些聚合操作，包括mean、count、min和sum等。你可能会猜想在你调用GroupBy对象的mean（）时发生了什么。很多常见的聚合，例如表10-1中的操作都有了优化实现。然而，你要用的可能并不局限于下面这个方法集。

表10-1：优化的groupby方法

函数名	描述
count	分组中的非 NA 值数量
sum	非 NA 值的累和
mean	非 NA 值的均值
median	非 NA 值的算术中位数
std、var	无偏的（n–1 分母）标准差和方差
min、max	非 NA 值的最小值、最大值
prod	非 NA 值的乘积
first、last	非 NA 值的第一个和最后一个值

你可以使用自行制定的聚合，并再调用已经在分组对象上定义好的方法。例如，你可能还记得quantile可以计算Series或DataFrame列的样本分位数。

尽管quantile并不是显式地为GroupBy对象实现的，但它是Series的方法，因此也可以用于聚合。在内部，GroupBy有效地对Series进行切片，为每一块调用piece.quantile（0.9），然后将这些结果一起组装到结果对象中：

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
```

```
Out[53]:
key1
a    1.668413
b   -0.523068
Name: data1, dtype: float64
```

要使用你自己的聚合函数，需要将函数传递给aggregate或agg方法：

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

你可能会注意到一些方法，比如describe也是有效的，尽管严格来说它们并不是聚合函数：

```
In [56]: grouped.describe()
Out[56]:
     data1
     count      mean       std       min       25%       50%       75
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943  1.2223
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 -0.5285
                data2
           max count      mean       std       min       25%       50
key1
a     1.965781   3.0  0.910916  0.712217  0.092908  0.669671  1.2464
b    -0.519439   2.0  0.525384  0.344556  0.281746  0.403565  0.5253
           75%       max
key1
a     1.319920  1.393406
b     0.647203  0.769023
```

我将在10.3节中更详细地解释此处发生了什么。

自定义聚合函数通常比表10-1中的优化函数慢得多。这是因为在构造中间组数据块时有一些额外的开销（函数调用、数据重新排列）

10.2.1 逐列及多函数应用

让我们回到之前例子中的小费数据集。在使用read_csv载入数据集后，我们增加一个小费比例列tip-pct：

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# 添加总账单的小费比例
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]

Out[59]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.059447
1       10.34  1.66     No  Sun  Dinner     3  0.160542
2       21.01  3.50     No  Sun  Dinner     3  0.166587
3       23.68  3.31     No  Sun  Dinner     2  0.139780
4       24.59  3.61     No  Sun  Dinner     4  0.146808
5       25.29  4.71     No  Sun  Dinner     4  0.186240
```

如你所见，对Series或DataFrame所有列进行聚合就是使用aggregate和所需函数，或者是调用像mean或std这种方法的。然而，你可能想根据各列同时使用多个函数进行聚合。幸运的是，这是可以做到的，我将会使用多个例子来阐明。首先，我将根据day和smoker来对tips进行分组：

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
```

请注意，要获得像表10-1中的描述性统计，你可以将函数名以字符串形式传递：

```
In [61]: grouped_pct = grouped['tip_pct']

In [62]: grouped_pct.agg('mean')
Out[62]:
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
```

```
     Yes       0.163863
Name: tip_pct, dtype: float64
```

如果你传递的是函数或者函数名的列表，你会获得一个列名是这些函数名的DataFrame：

```
In [63]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[63]:
                     mean       std  peak_to_peak
day  smoker
Fri  No          0.151650  0.028123      0.067349
     Yes         0.174783  0.051293      0.159925
Sat  No          0.158048  0.039767      0.235193
     Yes         0.147906  0.061375      0.290095
Sun  No          0.160113  0.042347      0.193226
     Yes         0.187250  0.154134      0.644685
Thur No          0.160298  0.038774      0.193350
     Yes         0.163863  0.039389      0.151240
```

这里我们传递了聚合函数的列表给agg方法，这些函数会各自运用于数据分组。

你可以不接受GroupBy对象给予各列的名称。请注意，lambda函数具有名称'<lambda>'，这使得它们难以识别（你可以查看函数的_name_属性）。因此，如果你传递的是（name，function）元组的列表，每个元组的第一个元素将作为DataFrame的列名（你可以认为二元元组的列表是一种有序的对应关系）：

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
                      foo       bar
day  smoker
Fri  No          0.151650  0.028123
     Yes         0.174783  0.051293
Sat  No          0.158048  0.039767
     Yes         0.147906  0.061375
Sun  No          0.160113  0.042347
     Yes         0.187250  0.154134
Thur No          0.160298  0.038774
     Yes         0.163863  0.039389
```

在DataFrame中，你有更多的选项，你可以指定应用到所有列上的函数列表或每一列上要应用的不同函数。假设我们想要计算tip_pct列和total_bill列的三个相同的统计值：

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [67]: result
Out[67]:
              tip_pct                     total_bill
                count      mean       max      count       mean    max
day  smoker
Fri  No             4  0.151650  0.187735          4  18.420000  22.75
     Yes           15  0.174783  0.263480         15  16.813333  40.17
Sat  No            45  0.158048  0.291990         45  19.661778  48.33
     Yes           42  0.147906  0.325733         42  21.276667  50.81
Sun  No            57  0.160113  0.252672         57  20.506667  48.17
     Yes           19  0.187250  0.710345         19  24.120000  45.35
Thur No            45  0.160298  0.266312         45  17.113111  41.19
     Yes           17  0.163863  0.241255         17  19.190588  43.11
```

如你所见，产生的DataFrame拥有分层列，与分别聚合每一列，再以列名作为keys参数使用concat将结果拼接在一起的结果相同：

```
In [68]: result['tip_pct']
Out[68]:
             count      mean       max
day  smoker
Fri  No          4  0.151650  0.187735
     Yes        15  0.174783  0.263480
Sat  No         45  0.158048  0.291990
     Yes        42  0.147906  0.325733
Sun  No         57  0.160113  0.252672
     Yes        19  0.187250  0.710345
Thur No         45  0.160298  0.266312
     Yes        17  0.163863  0.241255
```

和以前一样，可以传递具有自定义名称的元组列表：

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
                 tip_pct              total_bill
            Durchschnitt Abweichung Durchschnitt  Abweichung
day  smoker
Fri  No         0.151650   0.000791    18.420000   25.596333
     Yes        0.174783   0.002631    16.813333   82.562438
Sat  No         0.158048   0.001581    19.661778   79.908965
     Yes        0.147906   0.003767    21.276667  101.387535
```

```
Sun  No         0.160113   0.001793   20.506667   66.099980
     Yes        0.187250   0.023757   24.120000  109.046044
Thur No         0.160298   0.001503   17.113111   59.625081
     Yes        0.163863   0.001551   19.190588   69.808518
```

现在我们假设你想要将不同的函数应用到一个或多个列上。要实现这个功能，需要将含有列名与函数对应关系的字典传递给agg：

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[71]:
               tip  size
day  smoker
Fri  No       3.50     9
     Yes      4.73    31
Sat  No       9.00   115
     Yes     10.00   104
Sun  No       6.00   167
     Yes      6.50    49
Thur No       6.70   112
     Yes      5.00    40

In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....:              'size' : 'sum'})
Out[72]:
              tip_pct                                      size
                  min       max      mean       std  sum
day  smoker
Fri  No      0.120385  0.187735  0.151650  0.028123    9
     Yes     0.103555  0.263480  0.174783  0.051293   31
Sat  No      0.056797  0.291990  0.158048  0.039767  115
     Yes     0.035638  0.325733  0.147906  0.061375  104
Sun  No      0.059447  0.252672  0.160113  0.042347  167
     Yes     0.065660  0.710345  0.187250  0.154134   49
Thur No      0.072961  0.266312  0.160298  0.038774  112
     Yes     0.090014  0.241255  0.163863  0.039389   40
```

只有多个函数应用于至少一个列时，DataFrame才具有分层列。

10.2.2 返回不含行索引的聚合数据

在前面所有的例子中，聚合数据返回时都是带有索引的，有时索引是分层的，由唯一的分组键联合形成。因为不是所有的情况下都需要索引，所以在大多数情况下你可以通过向groupby传递as_index＝False来禁用分组键作为索引的行为：

```
In [73]: tips.groupby(['day', 'smoker'], as_index=False).mean()
Out[73]:
    day smoker  total_bill       tip      size   tip_pct
0   Fri     No   18.420000  2.812500  2.250000  0.151650
1   Fri    Yes   16.813333  2.714000  2.066667  0.174783
2   Sat     No   19.661778  3.102889  2.555556  0.158048
3   Sat    Yes   21.276667  2.875476  2.476190  0.147906
4   Sun     No   20.506667  3.167895  2.929825  0.160113
5   Sun    Yes   24.120000  3.516842  2.578947  0.187250
6  Thur     No   17.113111  2.673778  2.488889  0.160298
7  Thur    Yes   19.190588  3.030000  2.352941  0.163863
```

当然，通过在结果上调用reset_index也可以获得同样的结果。使用as_index＝False可以避免一些不必要的计算。

10.3 应用：通用拆分-应用-联合

GroupBy方法最常见的目的是apply（应用），这将是本节的内容。如图10-2所示，apply将对象拆分成多块，然后在每一块上调用传递的函数，之后尝试将每一块拼接到一起。

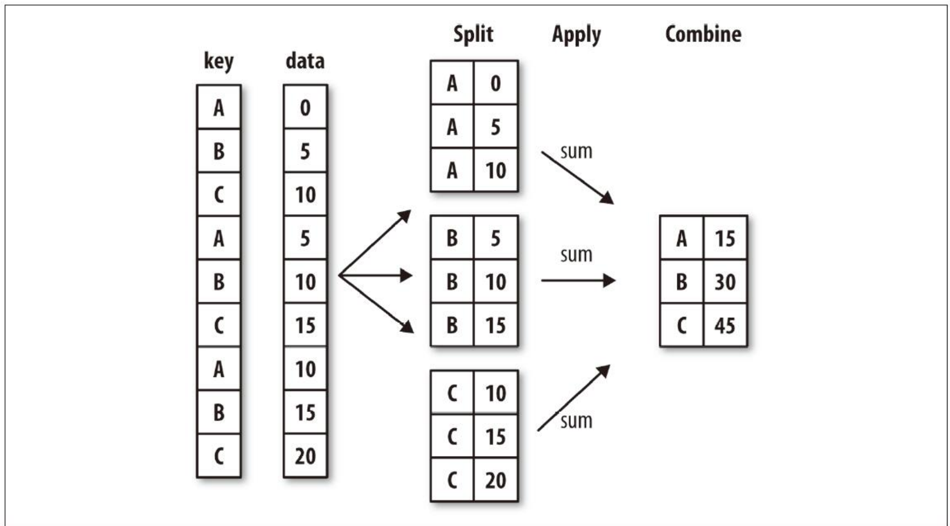


图10-2：分组聚合的阐述

回到之前的小费数据集，假设你想要按组选出小费百分比（tip-pct）最高的五组。首先，写一个可以在特定列中选出最大值所在行的函数：

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_pct
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
232       11.61  3.39     No  Sat  Dinner     2  0.291990
67         3.07  1.00    Yes  Sat  Dinner     1  0.325733
178        9.60  4.00    Yes  Sun  Dinner     2  0.416667
172        7.25  5.15    Yes  Sun  Dinner     2  0.710345
```

现在，如果我们按照smoker进行分组，之后调用apply，我们会得到

以下结果：

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     88        24.71  5.85     No  Thur   Lunch     2  0.236746
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       232       11.61  3.39     No   Sat  Dinner     2  0.291990
Yes    109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

这里发生了什么？top函数在DataFrame的每一行分组上被调用，之后使用pandas.concat将函数结果粘贴在一起，并使用分组名作为各组的标签。因此结果包含一个分层索引，该分层索引的内部层级包含原DataFrame的索引值：

如果你除了向apply传递函数，还传递其他参数或关键字的话，你可以把这些放在函数后进行传递：

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='tota
Out[77]:
                 total_bill    tip smoker   day    time  size   tip_
smoker day
No     Fri  94        22.75   3.25     No   Fri  Dinner     2  0.142
       Sat  212       48.33   9.00     No   Sat  Dinner     4  0.186
       Sun  156       48.17   5.00     No   Sun  Dinner     6  0.103
       Thur 142       41.19   5.00     No  Thur   Lunch     5  0.121
Yes    Fri  95        40.17   4.73    Yes   Fri  Dinner     4  0.117
       Sat  170       50.81  10.00    Yes   Sat  Dinner     3  0.196
       Sun  182       45.35   3.50    Yes   Sun  Dinner     3  0.077
       Thur 197       43.11   5.00    Yes  Thur   Lunch     4  0.115
```

除了基础的使用机制，还需要一些创造力才能充分地使用apply。传递函数的内部发生的事情取决于你自己，函数只需要返回一个pandas对象或一个标量值。本章的其余部分将包括介绍如何使用groupby解决多种问题的例子。

你可能还记得我之前在GroupBy对象上调用了describe方法：

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()

In [79]: result
Out[79]:
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.18
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.19
             max
smoker
No      0.291990
Yes     0.710345

In [80]: result.unstack('smoker')
Out[80]:
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

在GroupBy对象的内部，当你调用像describe这样的方法时，实际上是以下代码的简写：

```
f = lambda x: x.describe()
grouped.apply(f)
```

10.3.1　压缩分组键

在之前的例子中，你可以看到所得到的对象具有分组键所形成的分层索引以及每个原始对象的索引。你可以通过向groupby传递group_keys=False来禁用这个功能：

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
     total_bill   tip smoker   day    time  size   tip_pct
88        24.71  5.85     No  Thur   Lunch     2  0.236746
185       20.69  5.00     No   Sun  Dinner     5  0.241663
51        10.29  2.60     No   Sun  Dinner     2  0.252672
149        7.51  2.00     No  Thur   Lunch     2  0.266312
232       11.61  3.39     No   Sat  Dinner     2  0.291990
109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

10.3.2 分位数与桶分析

你可能会回忆起第8章中，pandas有一些工具，尤其是cut和qcut，用于将数据按照你选择的箱位或样本分位数进行分桶。与groupby方法一起使用这些函数可以对数据集更方便地进行分桶或分位分析。考虑一个简单的随机数据集和一个使用cut的等长桶分类：

```
In [82]: frame = pd.DataFrame({'data1': np.random.randn(1000),
   ....:                       'data2': np.random.randn(1000)})

In [83]: quartiles = pd.cut(frame.data1, 4)

In [84]: quartiles[:10]
Out[84]:
0      (-1.23, 0.489]
1     (-2.956, -1.23]
2      (-1.23, 0.489]
3      (0.489, 2.208]
4      (-1.23, 0.489]
5      (0.489, 2.208]
6      (-1.23, 0.489]
7      (-1.23, 0.489]
8      (0.489, 2.208]
9      (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, 0.489]
208] < (2.208, 3.928]]
```

cut返回的Categorical对象可以直接传递给groupby。所以我们可以计算出data2列的一个统计值集合，如下：

```
In [85]: def get_stats(group):
   ....:     return {'min': group.min(), 'max': group.max(),
   ....:             'count': group.count(), 'mean': group.mean()}

In [86]: grouped = frame.data2.groupby(quartiles)

In [87]: grouped.apply(get_stats).unstack()
Out[87]:
                 count       max      mean       min
data1
(-2.956, -1.23]   95.0  1.670835 -0.039521 -3.399312
(-1.23, 0.489]   598.0  3.260383 -0.002051 -2.989741
(0.489, 2.208]   297.0  2.954439  0.081822 -3.745356
(2.208, 3.928]    10.0  1.765640  0.024750 -1.929776
```

这些就是等长桶。为了根据样本分位数计算出等大小的桶，则需要使用qcut。我们将传递labels＝False来获得分位数数值：

```
# 返回分位数数值
In [88]: grouping = pd.qcut(frame.data1, 10, labels=False)

In [89]: grouped = frame.data2.groupby(grouping)

In [90]: grouped.apply(get_stats).unstack()
Out[90]:
       count       max      mean       min
data1
0      100.0  1.670835 -0.049902 -3.399312
1      100.0  2.628441  0.030989 -1.950098
2      100.0  2.527939 -0.067179 -2.925113
3      100.0  3.260383  0.065713 -2.315555
4      100.0  2.074345 -0.111653 -2.047939
5      100.0  2.184810  0.052130 -2.989741
6      100.0  2.458842 -0.021489 -2.223506
7      100.0  2.954439 -0.026459 -3.056990
8      100.0  2.735527  0.103406 -3.745356
9      100.0  2.377020  0.220122 -2.064111
```

我们将在第12章介绍pandas的Categorical类型。

10.3.3　示例：使用指定分组值填充缺失值

在清除缺失值时，有时你会使用dropna来去除缺失值，但是有时你可能想要使用修正值或来自于其他数据的值来输入（填充）到null值（NA）。fillna是一个可以使用的正确工具，例如这里我使用平均值来填充NA值：

```
In [91]: s = pd.Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0         NaN
1   -0.125921
2         NaN
3   -0.884475
4         NaN
5    0.227290
dtype: float64

In [94]: s.fillna(s.mean())
Out[94]:
0   -0.261035
1   -0.125921
2   -0.261035
3   -0.884475
4   -0.261035
5    0.227290
dtype: float64
```

假设你需要填充值按组来变化。一个方法是对数据分组后使用apply和一个在每个数据块上都调用fillna的函数。下面是一些将美国分为东部地区和西部地区的样本数据：

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   ....:           'Oregon', 'Nevada', 'California', 'Idaho']

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=states)

In [98]: data
Out[98]:
Ohio          0.922264
New York     -2.153545
```

```
Vermont      -0.365757
Florida      -0.375842
Oregon        0.329939
Nevada        0.981994
California    1.105913
Idaho        -1.613716
dtype: float64
```

请注意，语法['East']*4生成了一个包含四个['East']中元素的备份的列表。将列表拼接在一起。

让我们将数据中的一些值设置为缺失值：

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
Ohio          0.922264
New York     -2.153545
Vermont            NaN
Florida      -0.375842
Oregon        0.329939
Nevada             NaN
California    1.105913
Idaho              NaN
dtype: float64

In [101]: data.groupby(group_key).mean()
Out[101]:
East   -0.535707
West    0.717926
dtype: float64
```

我们使用分组的平均值来填充NA值，如下：

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.535707
Florida      -0.375842
Oregon        0.329939
Nevada        0.717926
California    1.105913
Idaho         0.717926
dtype: float64
```

在另一种情况下，你可能已经在代码中为每个分组预定义了填充值。由于每个分组都有一个内置的name属性，我们可以这样使用：

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio          0.922264
New York     -2.153545
Vermont       0.500000
Florida      -0.375842
Oregon        0.329939
Nevada       -1.000000
California    1.105913
Idaho        -1.000000
dtype: float64
```

10.3.4 示例：随机采样与排列

假设您想从大数据集中抽取随机样本（有或没有替换）以用于蒙特卡罗模拟目的或某些其他应用程序。有很多方法来执行“抽取”，这里我们使用Series的sample方法。

为了演示，这里我们讲解一种构造一副英式扑克牌的方法：

```
# 红桃，黑桃，梅花，方块
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)
deck = pd.Series(card_val, index=cards)
```

因此现在我们拥有了一个长度为52的Series，Series的索引包含了牌名，Series的值可以用于Blackjack和其他游戏（为了保持简单，我让'A'为1）：

```
In [108]: deck[:13]
Out[108]:
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

现在，基于我之前说的，从这副牌中拿出五张牌可以写成：

```
In [109]: def draw(deck, n=5):
   .....:     return deck.sample(n)
```

```
In [110]: draw(deck)
Out[110]:
AD     1
8C     8
5H     5
KC    10
2C     2
dtype: int64
```

假设你想要从每个花色中随机抽取两张牌。由于花色是牌名的最后两个字符，我们可以基于这点进行分组，并使用apply：

```
In [111]: get_suit = lambda card: card[-1] # last letter is suit

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
C  2C     2
   3C     3
D  KD    10
   8D     8
H  KH    10
   3H     3
S  2S     2
   4S     4
dtype: int64
```

或者我们也可以写成：

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
KC    10
JC    10
AD     1
5D     5
5H     5
6H     6
7S     7
KS    10
dtype: int64
```

10.3.5 示例：分组加权平均和相关性

在groupby的拆分-应用-联合的范式下，DataFrame的列间操作或两个Seriese之间的操作，例如分组加权平均是可以做到的。作为一个例子，我们使用一个包含分组键和权重值的数据集：

```
In [114]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
   .....:                                 'b', 'b', 'b', 'b'],
   .....:                    'data': np.random.randn(8),
   .....:                    'weights': np.random.rand(8)})

In [115]: df
Out[115]:
  category      data   weights
0        a  1.561587  0.957515
1        a  1.219984  0.347267
2        a -0.482239  0.581362
3        a  0.315667  0.217091
4        b -0.047852  0.894406
5        b -0.454145  0.918564
6        b -0.556774  0.277825
7        b  0.253321  0.955905
```

通过category进行分组加权平均如下：

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weig

In [118]: grouped.apply(get_wavg)
Out[118]:
category
a    0.811643
b   -0.122262
dtype: float64
```

作为另一个例子，考虑一个从雅虎财经上获得的数据集，该数据集包含一些标普500（SPX符号）和股票的收盘价：

```
In [119]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dat
   .....:                        index_col=0)

In [120]: close_px.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

一个有趣的任务可能是计算一个DataFrame，它包含标普指数（SPX）每日收益的年度相关性（通过百分比变化计算）。作为实现的一种方式，我们首先创建一个计算每列与'SPX'列成对关联的函数：

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

之后，我们使用pct_change计算close-px百分比的变化：

```
In [123]: rets = close_px.pct_change().dropna()
```

最后，我们按年对百分比变化进行分组，可以使用单行函数从每个行标签中提取每个datetime标签的year属性：

```
In [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
Out[126]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265  1.0
2004  0.374283  0.588531  0.557742  1.0
2005  0.467540  0.562374  0.631010  1.0
2006  0.428267  0.406126  0.518514  1.0
2007  0.508118  0.658770  0.786264  1.0
2008  0.681434  0.804626  0.828303  1.0
2009  0.707103  0.654902  0.797921  1.0
```

```
2010  0.710105  0.730118  0.839057  1.0
2011  0.691931  0.800996  0.859975  1.0
```

你也可以计算内部列相关性。这里我们计算了苹果和微软的年度相关性：

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

10.3.6　示例：逐组线性回归

在与前一个示例相同的主题中，只要该函数返回一个pandas对象或标量值，就可以使用groupby执行更复杂的按分组统计分析。例如，我可以定义以下regress（回归）函数（使用statsmodels计量经济学库），该函数对每个数据块执行普通最小二乘（OLS）回归：

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

现在，要计算AAPL在SPX回报上的年度线性回归，执行以下代码：

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
           SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

10.4　数据透视表与交叉表

数据透视表是电子表格程序和其他数据分析软件中常见的数据汇总工具。它根据一个或多个键聚合一张表的数据，将数据在矩形格式中排列，其中一些分组键是沿着行的，另一些是沿着列的。Python中的pandas透视表是通过本章所介绍的groupby工具以及使用分层索引的重塑操作实现的。DataFrame拥有一个pivot_table方法，并且还有还一个顶层的pandas.pivot_table函数。除了为groupby提供一个方便接口，pivot_table还可以添加部分总计，也称作边距。

回到小费数据集，假设你想要计算一张在行方向上按day和smoker排列的分组平均值（默认的pivot_table聚合类型）的表：

```
In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                 size       tip   tip_pct  total_bill
day  smoker
Fri  No      2.250000  2.812500  0.151650   18.420000
     Yes     2.066667  2.714000  0.174783   16.813333
Sat  No      2.555556  3.102889  0.158048   19.661778
     Yes     2.476190  2.875476  0.147906   21.276667
Sun  No      2.929825  3.167895  0.160113   20.506667
     Yes     2.578947  3.516842  0.187250   24.120000
Thur No      2.488889  2.673778  0.160298   17.113111
     Yes     2.352941  3.030000  0.163863   19.190588
```

这个功能也可以直接使用groupby实现。现在，假设我们只想在tip_pct和size上进行聚合，并根据time分组。我将把smoker放入表的列，而将day放入表的行：

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'
   .....:                  columns='smoker')
Out[131]:
                 size             tip_pct
smoker             No       Yes        No       Yes
time   day
Dinner Fri   2.000000  2.222222  0.139622  0.165347
       Sat   2.555556  2.476190  0.158048  0.147906
       Sun   2.929825  2.578947  0.160113  0.187250
       Thur  2.000000       NaN  0.159744       NaN
Lunch  Fri   3.000000  1.833333  0.187735  0.188937
       Thur  2.500000  2.352941  0.160311  0.163863
```

我们可以通过传递margins=True来扩充这个表来包含部分总计。这会添加All行和列标签，其中相应的值是单层中所有数据的分组统计值：

```
In [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'
   .....:                  columns='smoker', margins=True)
Out[132]:
                 size                               tip_pct
smoker             No       Yes       All        No       Yes       A
time   day
Dinner Fri   2.000000  2.222222  2.166667  0.139622  0.165347  0.158
       Sat   2.555556  2.476190  2.517241  0.158048  0.147906  0.153
       Sun   2.929825  2.578947  2.842105  0.160113  0.187250  0.166
       Thur  2.000000       NaN  2.000000  0.159744       NaN  0.159
Lunch  Fri   3.000000  1.833333  2.000000  0.187735  0.188937  0.188
       Thur  2.500000  2.352941  2.459016  0.160311  0.163863  0.161
All          2.668874  2.408602  2.569672  0.159328  0.163196  0.160
```

这里，All的值是均值，且该均值是不考虑吸烟者与非吸烟者（All列）或行分组中任何两级的（All行）。

要使用不同的聚合函数时，将函数传递给aggfunc。例如，'count'或者len将给出一张分组大小的交叉表（计数或出现频率）：

```
In [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], colu
   .....:                  aggfunc=len, margins=True)
Out[133]:
day             Fri   Sat   Sun  Thur    All
time   smoker
Dinner No       3.0  45.0  57.0   1.0  106.0
       Yes      9.0  42.0  19.0   NaN   70.0
Lunch  No       1.0   NaN   NaN  44.0   45.0
       Yes      6.0   NaN   NaN  17.0   23.0
All            19.0  87.0  76.0  62.0  244.0
```

如果某些情况下产生了空值（或者NA），你可能想要传递一个fill_value：

```
fill_value：
In [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker
   .....:                  columns='day', aggfunc='mean', fill_value=
Out[134]:
day                      Fri       Sat       Sun      Thur
time   size smoker
Dinner 1    No      0.000000  0.137931  0.000000  0.000000
            Yes     0.000000  0.325733  0.000000  0.000000
```

```
         2     No       0.139622  0.162705  0.168859  0.159744
               Yes      0.171297  0.148668  0.207893  0.000000
         3     No       0.000000  0.154661  0.152663  0.000000
               Yes      0.000000  0.144995  0.152660  0.000000
         4     No       0.000000  0.150096  0.148143  0.000000
               Yes      0.117750  0.124515  0.193370  0.000000
         5     No       0.000000  0.000000  0.206928  0.000000
               Yes      0.000000  0.106572  0.065660  0.000000
...                          ...       ...       ...       ...
Lunch    1     No       0.000000  0.000000  0.000000  0.181728
               Yes      0.223776  0.000000  0.000000  0.000000
         2     No       0.000000  0.000000  0.000000  0.166005
               Yes      0.181969  0.000000  0.000000  0.158843
         3     No       0.187735  0.000000  0.000000  0.084246
               Yes      0.000000  0.000000  0.000000  0.204952
         4     No       0.000000  0.000000  0.000000  0.138919
               Yes      0.000000  0.000000  0.000000  0.155410
         5     No       0.000000  0.000000  0.000000  0.121389
         6     No       0.000000  0.000000  0.000000  0.173706
[21 rows x 4 columns]
```

表10-2是pivot_table方法的概括。

表10-2：pivot_table选项

选项名	描述
values	需要聚合的列名；默认情况下聚合所有数值型的列
index	在结果透视表的行上进行分组的列名或其他分组键

表10-2：pivot_table选项（续）

选项名	描述
columns	在结果透视表的列上进行分组的列名或其他分组键
aggfunc	聚合函数或函数列表（默认情况下是 'mean'）；可是 groupby 上下文的任意有效函数
fill_value	在结果表中替换缺失值的值
dropna	如果为 True，将不含所有条目均为 NA 的列
margins	添加行 / 列小计和总计（默认为 False）

10.4.1 交叉表：crosstab

交叉表（简写为crosstab）是数据透视表的一个特殊情况，计算的是分组中的频率。下面是个例子：

```
In [138]: data
Out[138]:
   Sample Nationality    Handedness
0       1         USA  Right-handed
1       2       Japan   Left-handed
2       3         USA  Right-handed
3       4       Japan  Right-handed
4       5       Japan   Left-handed
5       6       Japan  Right-handed
6       7         USA  Right-handed
7       8         USA   Left-handed
8       9       Japan  Right-handed
9      10         USA  Right-handed
```

作为研究分析的一部分，我们可能想按照国籍和惯用性来总结这些数据。你可以使用pivot_table来实现这个功能，但是pandas.crosstable函数更为方便：

```
In [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True
Out[139]:
Handedness   Left-handed  Right-handed  All
Nationality
Japan                  2             3    5
USA                    1             4    5
All                    3             7   10
```

crosstab的前两个参数可是数组、Series或数组的列表。在小费数据中可以这么做：

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=Tru
Out[140]:
smoker        No  Yes  All
time   day
Dinner Fri     3    9   12
       Sat    45   42   87
       Sun    57   19   76
       Thur    1    0    1
Lunch  Fri     1    6    7
```

	Thur	44	17	61
All		151	93	244

10.5　本章小结

精通pandas的数据分组工具既可以帮助我们清洗数据，也对建模或统计分析工作有益。在第14章中，我们将在真实数据上介绍groupby的示例。

在下一章中，我们将把注意力转向时间序列数据。

第11章　时间序列

时间序列数据在很多领域都是重要的结构化数据形式，例如金融、经济、生态学、神经科学和物理学。在多个时间点观测或测量的数据形成了时间序列。许多时间序列是固定频率的，也就是说数据是根据相同的规则定期出现的，例如每15秒、每5分钟或每月1次。时间序列也可以是不规则的，没有固定的时间单位或单位间的偏移量。如何标记和引用时间序列数据取决于应用程序，并且您可能有以下其中一项：

·时间戳，具体的时刻。

·固定的时间区间，例如2007的1月或整个2010年。

·时间间隔，由开始和结束时间戳表示。时间区间可以被认为是间隔的特殊情况。

·实验时间或消耗时间。每个时间戳是相对于特定开始时间的时间的量度（例如，自从被放置在烤箱中每秒烘烤的饼干的直径）。

在本章中，我主要关注前三类中的时间序列，尽管许多技术可以应用于实验时间序列，其中索引可以是整数，也可以是从实验开始时的消耗时间的浮点数。最简单和最广泛使用的时间序列是那些由时间戳索引的。

pandas也支持基于时间间隔的索引，这是一种表示实验时间或消耗时间的有效方式。我们在本书中将不会探索时间间隔索引，但是你可以在pandas官方文档（http://pandas.pydata.org ）中学会更多。

11.1　日期和时间数据的类型及工具

Python标准库包含了日期和时间数据的类型，也包括日历相关的功能。datetime、time和calendar模块是开始处理时间数据的主要内容。datetime.datetime类型，或简写为datetime，是广泛使用的：

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

datetime既存储了日期，也存储了细化到微秒的时间。timedelta表示两个datetime对象的时间差：

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

你可以为一个datetime对象加上（或减去）一个timedelta或其整数倍来产生一个新的datetime对象：

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)

In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

表11-1概括了datetime模块的数据类型。尽管本章主要关注的是pandas中的数据类型和高阶时间序列操作，但你仍然可能在其他地方遇到基于datetime的类型。

表11-1：datetime模块中的类型

类型	描述
date	使用公历日历存储日历日期（年，月，日）
time	将时间存储为小时，分钟，秒和微秒
datetime	存储日期和时间
timedelta	表示两个 datetime 值之间的差（如日，秒和微秒）
tzinfo	用于存储时区信息的基本类型

11.1.1 字符串与datetime互相转换

你可以使用str方法或传递一个指定的格式给strftime方法来对datetime对象和pandas的Timestamp对象进行格式化，这些我将在稍后介绍：

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

表11-2是格式代码的完整列表（从第2章中复现）。

表11-2：datetime格式说明（兼容ISO C89）

类型	描述
%Y	四位的年份
%y	两位的年份
%m	两位的月份 [01, 12]
%d	两位的日期号 [01, 31]
%H	小时，24 小时制 [00, 23]
%I	小时，12 小时制 [01, 12]
%M	两位的分钟 [00, 59]
%S	秒 [00, 61]（60、61 是闰秒）
%w	星期日期 [0（星期天），6]
%U	一年中的星期数 [00, 53]。以星期天为每周的第一天，一年中第一个星期天前的日期作为 " 第 0 周 "
%W	一年中的星期数 [00, 53]。以星期一为每周的第一天，一年中第一个星期一前的日期作为 " 第 0 周 "
%z	格式为 +HHMM 或 -HHMM 的 UTC 时区偏移；如果没有时区则为空
%F	%Y-%m-%d 的简写（例如，2012-4-18）
%D	%m/%d/%y 的简写（例如，04/18/12）

你可以使用datetime.srtptime和这些格式代码，将字符串转换日期：

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

datetime.strptime是在已知格式的情况下转换日期的好方式。然而，每次都必须编写一个格式代码可能有点烦人，特别是对于通用日期格式。在这种情况下，你可以使用第三方dateutil包的parser.parse方法（这个包在安装pandas时已经自动安装）：

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil能够解析大部分人类可理解的日期表示：

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

在国际场合下，日期出现在月份之前很常见，因此你可以传递dayfirst＝True来表明这种情况：

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas主要是面向处理日期数组的，无论是用作轴索引还是用作DataFrame中的列。to_datetime方法可以转换很多不同的日期表示格式。标准日期格式，比如ISO 8601可以非常快地转换：

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'
etime64[ns]', freq=None)
```

to_datetime方法还可以处理那些被认为是缺失值的值（None、空字符串等）：

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'
pe='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False,  True], dtype=bool)
```

NaT（Not a time）是pandas中时间戳数据的是null值。

dateutil.parser是一个有用但并不完美的工具。值得注意的是，它会将一些字符串识别为你并不想要的日期——例如，'42'将被解析为2042年的当前日期。

datetime对象还拥有许多其他国家或语言系统的本地化格式选项。例如，德语或法语的月份名称会和英语系统有所不同。参见表11-3的选项列表。

表11-3：特定地区日期格式化选项

类型	描述
%a	缩写的工作日名称
%A	全写的工作日名称
%b	简写的月份名称
%B	全写的月份名称
%c	完整的日期和时间（例如，‘Tue 01 May 2012 04:20:57 PM’）
%p	AM 或 PM 的地区等效
%x	适合地区的格式化日期（例如，在美国，May 1, 2012 会生成 '05 / 01/2012'）
%X	适合地区的时间（例如，’ 04:24:12 PM’）

11.2 时间序列基础

pandas中的基础时间序列种类是由时间戳索引的Series，在pandas外部则通常表示为Python字符串或datetime对象：

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....:          datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....:          datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

在这种情况下，这些datetime对象可以被放入DatetimeIndex中：

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

和其他Series类似，不同索引的时间序列之间的算术运算在日期上自动对齐：

```
In [44]: ts + ts[::2]
Out[44]:
2011-01-02   -0.409415
2011-01-05         NaN
2011-01-07   -1.038877
2011-01-08         NaN
2011-01-10    3.931561
2011-01-12         NaN
dtype: float64
```

ts[：：2]会将ts中每隔一个的元素选择出。

pandas使用NumPy的datetime64数据类型在纳秒级的分辨率下存储时间戳：

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

DatetimeIndex中的标量值是pandas的Timestamp对象：

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

所有使用datetime对象的地方都可以用Timestamp。此外，Timestamp还可以存储频率信息（如果有的话）并了解如何进行时区转换和其他类型操作。这些处理会稍后介绍。

11.2.1　索引、选择、子集

当你基于标签进行索引和选择时，时间序列的行为和其他的pandas.Series类似：

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.5194387150567381l
```

为了方便，你还可以传递一个能解释为日期的字符串：

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

对一个长的时间序列，可以传递一个年份或一个年份和月份来轻松地选择数据的切片：

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....:                              index=pd.date_range('1/1/2000', perioc

In [53]: longer_ts
Out[53]:
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
2000-01-06   -1.296221
2000-01-07    0.274992
2000-01-08    0.228913
2000-01-09    1.352917
2000-01-10    0.886429
                 ...
2002-09-17   -0.139298
2002-09-18   -1.159926
2002-09-19    0.618965
2002-09-20    1.373890
2002-09-21   -0.983505
2002-09-22    0.930944
2002-09-23   -0.811676
```

```
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:
2001-01-01    1.599534
2001-01-02    0.474071
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
2001-01-06    0.106403
2001-01-07   -1.308228
2001-01-08    2.173185
2001-01-09    0.564561
2001-01-10   -0.190481
                 ...
2001-12-22    0.000369
2001-12-23    0.900885
2001-12-24   -0.454869
2001-12-25   -0.864547
2001-12-26    1.129120
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

这里，字符串'2001'被解释为一个年份，并选择了相应的时间区间。如果你指定了月份也是有效的：

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01   -0.622547
2001-05-02    0.936289
2001-05-03    0.750018
2001-05-04   -0.056715
2001-05-05    2.300675
2001-05-06    0.569497
2001-05-07    1.489410
2001-05-08    1.264250
2001-05-09   -0.761837
2001-05-10   -0.331617
                 ...
2001-05-22    0.503699
2001-05-23   -1.387874
2001-05-24    0.204851
2001-05-25    0.603705
2001-05-26    0.545680
```

```
2001-05-27    0.235477
2001-05-28    0.111835
2001-05-29   -1.251504
2001-05-30   -2.949343
2001-05-31    0.634634
Freq: D, Length: 31, dtype: float64
```

使用datetime对象进行切片也是可以的：

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

因为大部分的时间序列数据是按时间顺序排序的，你可以使用不包含在时间序列中的时间戳进行切片，以执行范围查询：

```
In [57]: ts
Out[57]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
dtype: float64
```

和之前一样，你可以传递一个字符串的日期、datetime对象或者时间戳。请记住通过这种方式的切片产生了原时间序列的视图，类似于NumPy的数组。这意味着没有数据被复制，并且在切片上的修改会反映在原始数据上。

有一个等价实例方法，truncate，它可以在两个日期间对Series进行切片：

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
dtype: float64
```

上面这些操作也都适用于DataFrame，并在其行上进行索引：

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
   ....:                        index=dates,
   ....:                        columns=['Colorado', 'Texas',
   ....:                                 'New York', 'Ohio'])

In [62]: long_df.loc['5-2001']
Out[62]:
            Colorado     Texas  New York      Ohio
2001-05-02 -0.006045  0.490094 -0.277186 -0.707213
2001-05-09 -0.560107  2.735527  0.927335  1.513906
2001-05-16  0.538600  1.273768  0.667876 -0.969206
2001-05-23  1.676091 -0.817649  0.050188  1.951312
2001-05-30  3.260383  0.963301  1.201206 -1.852001
```

11.2.2　含有重复索引的时间序列

在某些应用中，可能会有多个数据观察值落在特定的时间戳上。下面是个例子：

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000
   ....:                           '1/2/2000', '1/3/2000'])

In [64]: dup_ts = pd.Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

通过检查索引的is_unique属性，我们可以看出索引并不是唯一的：

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

对上面的Series进行索引，结果是标量值还是Series切片取决于是否有时间戳是重复的：

```
In [67]: dup_ts['1/3/2000']  # 不重复
Out[67]: 4

In [68]: dup_ts['1/2/2000']  # 重复
Out[68]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

假设你想要聚合含有非唯一时间戳的数据。一种方式就是使用groupby并传递level=0：

```
In [69]: grouped = dup_ts.groupby(level=0)
```

```
In [70]: grouped.mean()
Out[70]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

11.3　日期范围、频率和移位

pandas的通用时间序列是不规则的，即时间序列的频率不是固定的。对于很多应用来说，这足够了。然而，经常有需要处理固定频率的场景，例如每日的、每月的或每15分钟，这意味着我们甚至需要在必要的时候向时间序列中引入缺失值。幸运的是，pandas拥有一整套标准的时间序列频率和工具用于重新采样、推断频率以及生成固定频率的数据范围。例如，你可以通过调用resample方法将样本时间序列转换为固定的每日频率数据：

```
In [72]: ts
Out[72]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

字符串'D'被解释为每日频率。

在频率间转换，又称为重新采样，是一个足够大的话题，将会在稍后拥有独自一节（11.6节）。这里我将向你展示如何使用基础频率及其倍数。

11.3.1　生成日期范围

尽管我之前使用的时候没有解释，pandas.date_range是用于根据特定频率生成指定长度的DatetimeIndex：

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

默认情况下，date_range生成的是每日的时间戳。如果你只传递一个起始或结尾日期，你必须传递一个用于生成范围的数字：

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01
              dtype='datetime64[ns]', freq='D')
```

开始日期和结束日期严格定义了生成日期索引的边界。例如，如果你需要一个包含每月最后业务日期的时间索引，你可以传递'BM'频率（business end of month，月度业务结尾；参考表11-4的频率列表），只有落在或在日期范围内的日期会被包括：

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

表11-4：基础时间序列频率（不全）

别名	偏置类型	描述
D	Day	日历日的每天
B	BusinessDay	工作日的每天
H	Hour	每小时
T 或 min	Minute	每分钟
S	Second	每秒
L 或 ms	Milli	每毫秒（1/1,000 秒)
U	Micro	每微秒（1/1,000,000 秒)
M	MonthEnd	日历日的月底日期
BM	BusinessMonthEnd	工作日的月底日期
MS	MonthBegin	日历日的月初日期
BMS	BusinessMonthBegin	工作日的月初日期
W-MON, W-TUE, ...	Week	按照给定的星期日期按每周取日期（MON, TUE, WED, THU, FRI, SAT 或 SUN)
WOM-1MON,WOM-2MON,...	WeekOfMonth	在本月的第一,二,三或四周创建按周分隔的日期（例如 WOM-3FRI 代表每月的第三个星期五）
Q-JAN, Q-FEB, ...	QuarterEnd	每个月最后一个日历日的季度

表11-4：基础时间序列频率（不全）（续）

别名	偏置类型	描述
		日期，以表示月份结束的年份(JAN，FEB,MAR，APR，MAY，JUN, JUL, AUG, SEP, OCT, NOV 或 DEC)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	每个月最后一个工作日对应的季度日期，以表示月份结束的年份
QS-JAN, QS-FEB, ...	QuarterBegin	每个月第一个日历日对应的季度日期，以表示月份结束的年份
BQS-JAN, BQS-FEB, ...	YearEnd	给定月份所在的月的最后一个日历日所对应的年度日期
A-JAN, A-FEB, ...	BusinessYearEnd	给定月份所在的月的最后一个工作日所对应的年度日期（JAN，FEB，MAR，APR，MAY，JUN，JUL，AUG，SEP，OCT，NOV 或 DEC)
BA-JAN, BA-FEB, ...	YearBegin	给定月份所在的月的第一个日历日所对应的年度日期
AS-JAN, AS-FEB, ...	BusinessYearBegin	给定月份所在的月的第一个工作日所对应的年度日期
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	给定月份所在月的第一个工作日所对应的年度日期

默认情况下，date_range保留开始或结束时间戳的时间（如果有的话）：

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

有时候你会获得包含时间信息的开始日期或结束日期，但是你想要生成的是标准化为零点的时间戳。有一个normalize选项可以实现这个功能：

11.3.2　频率和日期偏置

pandas中的频率是由基础频率和倍数组成的。基础频率通常会有字符串别名，例如'M'代表每月，'H'代表每小时。对于每个基础频率，都有一个对象可以被用于定义日期偏置。例如，每小时的频率可以使用Hour类来表示：

```
In [81]: from pandas.tseries.offsets import Hour, Minute

In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

你可以传递一个整数来定义偏置量的倍数：

```
In [84]: four_hours = Hour(4)

In [85]: four_hours
Out[85]: <4 * Hours>
```

在大多数应用中，你都不需要显式地创建这些对象，而是使用字符串别名，如'H'或'4H'。在基础频率前放一个整数就可以生成倍数：

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

多个偏置可以通过加法进行联合：

```
In [87]: Hour(2) + Minute(30)
```

```
Out[87]: <150 * Minutes>
```

类似地，你可以传递频率字符串，例如'1h30min'将会有效地转换为同等的表达式：

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

有些频率描述点的时间并不是均匀分隔的。例如，'M'（日历月末）和'BM'（月内最后工作日）取决于当月天数，以及像之后的例子中，取决于月末是否是周末。我们将这些日期称为锚定偏置量。

根据表11-4，可以获得pandas中的频率代码和日期偏置类。

用户可以自行定义频率类型，用于提供pandas中没有的日期逻辑，但完整的实现细节是超出本书内容的。

11.3.2.1 月中某星期的日期

"月中某星期"（week of month）的日期是一个有用的频率类，以'WOM'开始。它允许你可以获取每月第三个星期五这样的日期：

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

11.3.3　移位（前向和后向）日期

"移位"是指将日期按时间向前移动或向后移动。Series和DataFrame都有一个shift方法用于进行简单的前向或后向移位，而不改变索引：

```
In [91]: ts = pd.Series(np.random.randn(4),
   ....:                index=pd.date_range('1/1/2000', periods=4, f

In [92]: ts
Out[92]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

当我们像上面这样进行移位时，会在时间序列的起始位或结束位引入缺失值。

shift常用于计算时间序列或DataFrame多列时间序列的百分比变化，代码实现如下：

```
ts / ts.shift(1) - 1
```

由于简单移位并不改变索引，一些数据会被丢弃。因此，如果频率是已知的，则可以将频率传递给shift来推移时间戳而不是简单的数据：

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
```

其他的频率也可以传递，为你前移和后移数据提供灵活性：

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03   -0.066748
2000-03-03    0.838639
2000-04-03   -0.117388
2000-05-03   -0.517795
dtype: float64

In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00   -0.066748
2000-02-29 01:30:00    0.838639
2000-03-31 01:30:00   -0.117388
2000-04-30 01:30:00   -0.517795
Freq: M, dtype: float64
```

这里的T代表分钟。

11.3.3.1　使用偏置进行移位日期

pandas日期偏置也可以使用datetime或Timestamp对象完成：

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

如果你添加了一个锚定偏置量，比如MonthEnd，根据频率规则，第一个增量会将日期“前滚”到下一个日期：

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')
```

```
In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

锚定偏置可以使用rollforward和rollback分别显式地将日期向前或向后"滚动"：

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

将移位方法与groupby一起使用是日期偏置的一种创造性用法：

```
In [106]: ts = pd.Series(np.random.randn(20),
   .....:                index=pd.date_range('1/15/2000', periods=20
In [107]: ts
Out[107]:
2000-01-15   -0.116696
2000-01-19    2.389645
2000-01-23   -0.932454
2000-01-27   -0.229331
2000-01-31   -1.140330
2000-02-04    0.439920
2000-02-08   -0.823758
2000-02-12   -0.520930
2000-02-16    0.350282
2000-02-20    0.204395
2000-02-24    0.133445
2000-02-28    0.327905
2000-03-03    0.072153
2000-03-07    0.131678
2000-03-11   -1.297459
2000-03-15    0.997747
2000-03-19    0.870955
2000-03-23   -0.991253
2000-03-27    0.151699
2000-03-31    1.266151
Freq: 4D, dtype: float64
In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
dtype: float64
```

当然，使用resample是更简单更快捷的方法（我们将在11.6节中深入讨论resample）：

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
Freq: M, dtype: float64
```

11.4 时区处理

处理时区通常是时间序列操作中最不愉快的部分。因此，很多时间序列用户选择世界协调时间或UTC，它是格林尼治时间的后继者，也是目前的国际标准。时区通常被表示为UTC的偏置，例如，在夏令时期间，纽约比UTC时间晚4个小时，其余时间晚5个小时。

在Python语言中，时区信息来源于第三方库pytz（可以使用pip或conda安装），其中公开了Olson数据库，这是世界时区信息的汇编。这对于历史数据尤为重要，因为夏令时（DST）转换日期（甚至是UTC偏移量）已经根据地方政府的意愿而改变了很多次。在美国，DST转换时间从1900至今已经多次变更！

如果需要关于pytz库的更多细节信息，你需要查看该库的官方文档。本书目前的内容中，pandas封装了pytz的功能，因此你可以忽略pytz时区名称以外的API。时区名称可以在文档中找到：

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'U
```

要获得pytz的时区对象，可使用pytz.timezone：

```
In [112]: tz = pytz.timezone('America/New_York')

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

pandas中的方法可以接收时区名称或时区对象。

11.4.1　时区的本地化和转换

默认情况下，pandas中的时间序列是时区简单型的。例如，考虑下面的时间序列：

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')

In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [116]: ts
Out[116]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64
```

索引的tz属性是None：

```
In [117]: print(ts.index.tz)
None
```

日期范围可以通过时区集合来生成：

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='U
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:0
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:0
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:0
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:0
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:0
              dtype='datetime64[ns, UTC]', freq='D')
```

使用tz_localize方法可以从简单时区转换到本地化时区：

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
```

```
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00   -0.202469
2012-03-10 09:30:00+00:00    0.050718
2012-03-11 09:30:00+00:00    0.639869
2012-03-12 09:30:00+00:00    0.597594
2012-03-13 09:30:00+00:00   -0.797246
2012-03-14 09:30:00+00:00    0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:0
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:0
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:0
              dtype='datetime64[ns, UTC]', freq='D')
```

一旦时间序列被本地化为某个特定的时区，则可以通过tz_convert将其转换为另一个时区：

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00   -0.202469
2012-03-10 04:30:00-05:00    0.050718
2012-03-11 05:30:00-04:00    0.639869
2012-03-12 05:30:00-04:00    0.597594
2012-03-13 05:30:00-04:00   -0.797246
2012-03-14 05:30:00-04:00    0.472879
Freq: D, dtype: float64
```

在之前的跨越America/New-York时区的DST转换时间序列的例子中，我们可以将其本地化到EST并转换为UTC或柏林时间：

```
In [124]: ts_eastern = ts.tz_localize('America/New_York')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00   -0.202469
2012-03-10 14:30:00+00:00    0.050718
2012-03-11 13:30:00+00:00    0.639869
2012-03-12 13:30:00+00:00    0.597594
```

```
2012-03-13 13:30:00+00:00   -0.797246
2012-03-14 13:30:00+00:00    0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00   -0.202469
2012-03-10 15:30:00+01:00    0.050718
2012-03-11 14:30:00+01:00    0.639869
2012-03-12 14:30:00+01:00    0.597594
2012-03-13 14:30:00+01:00   -0.797246
2012-03-14 14:30:00+01:00    0.472879
Freq: D, dtype: float64
```

tz_localize和tz_convert也是DatetimeIndex的实例方法：

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:0
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:0
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:0
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

 简单时间戳的本地化也会检查夏时制转换周围的模糊或不存在的时间。

11.4.2　时区感知时间戳对象的操作

与时间序列和日期范围类似，单独的Timestamp对象也可以从简单时间戳本地化为时区感知时间戳，并从一个时区转换为另一个时区：

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')

In [129]: stamp_utc = stamp.tz_localize('utc')

In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York
```

你也可以在创建Timestamp的时候传递一个时区：

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe

In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

时区感知的Timestamp对象内部存储了一个Unix纪元（1970年1月1日）至今的纳秒数量UTC时间戳数值，该数值在时区转换中是不变的：

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 1299902400000000000
```

在使用pandas的DateOffset进行时间算术时，pandas尽可能遵从夏时制。这里我们构建恰好在DST转换之前发生的时间戳（向前和向后）。首先，我们构造转换到DST之前的30分钟的时间：

```
In [135]: from pandas.tseries.offsets import Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')

In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')

In [138]: stamp + Hour()
```

```
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

之后，我们构建从DST进行转换前的90分钟：

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')
In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [141]: stamp + 2 * Hour()
Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

11.4.3　不同时区间的操作

如果两个时区不同的时间序列需要联合，那么结果将是UTC时间的。由于时间戳以UTC格式存储，这是一个简单的操作，不需要转换：

```
In [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts
Out[144]:
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
2012-03-15 09:30:00   -0.922839
2012-03-16 09:30:00    0.312656
2012-03-19 09:30:00   -1.128497
2012-03-20 09:30:00   -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')

In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index

Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5　时间区间和区间算术

时间区间表示的是时间范围，比如一些天、一些月、一些季度或者是一些年。Period类表示的正是这种数据类型，需要一个字符串或数字以及表11-4中的频率：

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

在这个例子中，Period对象表示的是从2007年1月1日到2007年12月31日（包含在内）的时间段。在时间段上增加或减去整数可以方便地根据它们的频率进行移位。

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

如果两个区间拥有相同的频率，则它们的差是它们之间的单位数：

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

使用period_range函数可以构造规则区间序列：

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M'

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2
00-06'], dtype='period[M]', freq='M')
```

PeriodIndex类存储的是区间的序列，可以作为任意pandas数据结构的轴索引：

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01   -0.514551
2000-02   -0.559782
2000-03   -0.783408
2000-04   -1.797685
2000-05   -0.172670
2000-06    0.680215
Freq: M, dtype: float64
```

如果你有一个字符串数组，你也可以使用PeriodIndex类：

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[
='Q-DEC')
```

11.5.1 区间频率转换

使用asfreq可以将区间和PeriodIndex对象转换为其他的频率。例如，假设我们有一个年度区间，并且想要在一年的开始或结束时将其转换为月度区间。这非常简单：

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

你可以将Period（'2007'，'A-DEC'）看作一段时间中的一种游标，将时间按月份划分，参见图11-1。对于除十二月以外的一个月结束的财政年度，相应的每月分期是不同的：

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

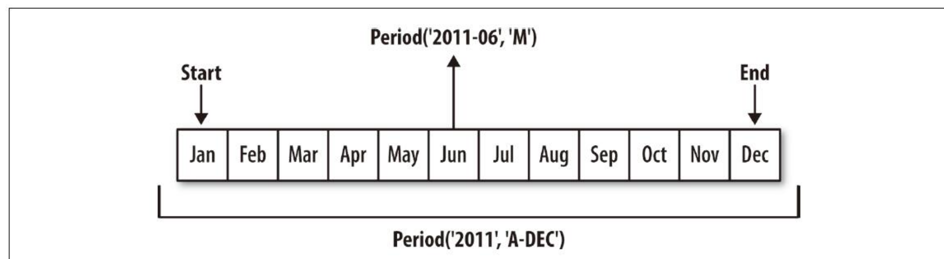


图11-1：区间频率转换示意图

当你从高频率向低频率转换时，pandas根据子区间的"所属"来决定父区间。例如，在A-JUN频率中，Aug-2007是2008区间的一部分：

```
In [168]: p = pd.Period('Aug-2007', 'M')

In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

完整的PeriodIndex对象或时间序列可以按照相同的语义进行转换：

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [172]: ts
Out[172]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01    1.607578
2007-01    0.200381
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64
```

这里，年度区间将被替换为对应于每个年度区间内的第一个月的月度区间。如果我们想要每年最后一个工作日，我们可以使用'B'频率来表示我们想要的是区间的末端：

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

11.5.2 季度区间频率

季度数据是会计、金融和其他领域的标准。很多季度数据是在财年结尾报告的，通常是一年12个月中的最后一个日历日或工作日。因此，由于是财年结尾，区间2012Q4有着不同的意义。pandas支持所有的可能的12个季度频率从Q-JAN到Q-DEC：

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

在财年结束于1月的情况下，2012Q4运行时间为11月至1月，你可以通过转换为每日频率进行检查，参见图11-2。

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```

因此，做简单的区间算术是可行的，例如，要获取在季度倒数第二个工作日下午4点的时间戳，你可以这么做：

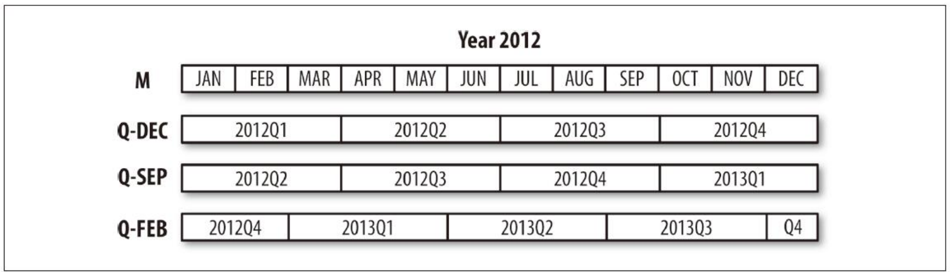


图11-2：不同的季度频率约定

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')
```

```
In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

你可以使用peroid_range生成季度序列。它的算术也是一样的：

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16

In [186]: ts.index = new_rng.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

11.5.3　将时间戳转换为区间（以及逆转换）

通过时间戳索引的Series和DataFrame可以被to_period方法转换为区间：

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')

In [189]: ts = pd.Series(np.random.randn(3), index=rng)

In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()

In [192]: pts
Out[192]:
2000-01    1.663261
2000-02   -0.996206
2000-03    1.521760
Freq: M, dtype: float64
```

由于区间是非重叠时间范围，一个时间戳只能属于给定频率的单个区间。尽管默认情况下根据时间戳推断出新PeriodIndex的频率，但你可以指定任何想要的频率。在结果中包含重复的区间也是没有问题的：

```
In [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [195]: ts2
Out[195]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
Out[196]:
2000-01    0.244175
```

```
2000-01    0.423331
2000-01   -0.654040
2000-02    2.089154
2000-02   -0.060220
2000-02   -0.167933
Freq: M, dtype: float64
```

使用to_timestamp可以将区间再转换为时间戳：

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64
```

11.5.4 从数组生成PeriodIndex

固定频率数据集有时存储在跨越多列的时间范围信息中。例如，在这个宏观经济数据集中，年份和季度在不同列中：

```
In [200]: data = pd.read_csv('examples/macrodata.csv')

In [201]: data.head(5)
Out[201]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19

In [202]: data.year
Out[202]:
0      1959.0
1      1959.0
2      1959.0
3      1959.0
4      1960.0
5      1960.0
6      1960.0
7      1960.0
8      1961.0
9      1961.0
        ...
193    2007.0
194    2007.0
195    2007.0
196    2008.0
197    2008.0
198    2008.0
199    2008.0
200    2009.0
201    2009.0
202    2009.0
Name: year, Length: 203, dtype: float64

In [203]: data.quarter
Out[203]:
0      1.0
```

```
1      2.0
2      3.0
3      4.0
4      1.0
5      2.0
6      3.0
7      4.0
8      1.0
9      2.0
      ...
193    2.0
194    3.0
195    4.0
196    1.0
197    2.0
198    3.0
199    4.0
200    1.0
201    2.0
202    3.0
Name: quarter, Length: 203, dtype: float64
```

通过将这些数组和频率传递给PeriodIndex，你可以联合这些数组形成DataFrame的索引：

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarte
   .....:                        freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index
In [207]: data.infl
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
1960Q2    0.14
1960Q3    2.70
1960Q4    1.21
1961Q1   -0.40
1961Q2    1.47
          ...
```

```
2007Q2    2.75
2007Q3    3.45
2007Q4    6.38
2008Q1    2.82
2008Q2    8.53
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 重新采样与频率转换

重新采样是指将时间序列从一个频率转换为另一个频率的过程。将更高频率的数据聚合到低频率被称为向下采样，而从低频率转换到高频率称为向上采样。并不是所有的重新采样都属于上面说的两类；例如，将W-WED（weekly on Wednesday，每周三）转换到W-FRI（每周五）既不是向上采样也不是向下采样。

pandas对象都配有resample方法，该方法是所有频率转换的工具函数。resample拥有类似于groupby的API；你调用resample对数据分组，之后再调用聚合函数：

```
In [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')

In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [210]: ts
Out[210]:
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
2000-01-06   -0.024330
2000-01-07   -2.047939
2000-01-08   -0.272657
2000-01-09   -1.692615
2000-01-10    1.423830
                 ...
2000-03-31   -0.007852
2000-04-01   -1.638806
2000-04-02    1.401227
2000-04-03    1.758539
2000-04-04    0.628932
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-01-31   -0.165893
2000-02-29    0.078606
2000-03-31    0.223811
2000-04-30   -0.063643
Freq: M, dtype: float64
```

```
In [212]: ts.resample('M', kind='period').mean()
Out[212]:
2000-01   -0.165893
2000-02    0.078606
2000-03    0.223811
2000-04   -0.063643
Freq: M, dtype: float64
```

resample是一个灵活且高性能的方法，可以用于处理大型时间序列，下一节的例子将会说明它的语义和用途。表11-5总结了resample方法的部分选项。

表11-5：resample方法参数

参数	描述
freq	表明所需采样频率的字符串或 DateOffset 对象（例如，'M'、'5min' 或 Second(1)）
axis	需要采样的轴向；默认是 axis=0
fill_method	向上采样时的插值方式，'ffill' 或 'bfill'；默认是不插值的
closed	向下采样中，每段间隔的哪一段是封闭的（包含的），'right' 或 'left'
label	向下采样中，如何用 'right' 或 'left' 的箱标签标记聚合结果（例如，9:30 到 9:35 的五分钟间隔可以被标记为 9:30 或 9:35）
loffset	对箱标签进行时间调校，例如 '-1s'/Second（-1）可以将聚合标签向前移动一秒
limit	在前向或后向填充时，填充区间的最大值
kind	对区间（'period'）或时间戳（'timestamp'）的聚合；默认为时间序列索引的类型
convention	在对区间重新采样时，用于将低频周期转换为高频的约定（'start' 或 'end'）；默认为 'end'

11.6.1 向下采样

将数据聚合到一个规则的低频率上是一个常见的时间序列任务。你要聚合的数据不必是固定频率的。期望的频率定义了用于对时间序列切片以聚合的箱体边界。例如，要将时间转换为每月，'M'或'BM'，你需要将数据分成一个月的时间间隔。每个间隔是半闭合的，一个数据点只能属于一个时间间隔，时间间隔的并集必须是整个时间帧。在使用resample进行向下采样数据时有些事情需要考虑：

·每段间隔的哪一边是闭合的

·如何在间隔的起始或结束位置标记每个已聚合的箱体

为了说明，让我们看下一些一分钟内的数据：

```
In [213]: rng = pd.date_range('2000-01-01', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts
Out[215]:
2000-01-01 00:00:00     0
2000-01-01 00:01:00     1
2000-01-01 00:02:00     2
2000-01-01 00:03:00     3
2000-01-01 00:04:00     4
2000-01-01 00:05:00     5
2000-01-01 00:06:00     6
2000-01-01 00:07:00     7
2000-01-01 00:08:00     8
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: T, dtype: int64
```

假设你想通过计算每一组的加和将这些数据聚合到五分钟的块或柱内：

```
In [216]: ts.resample('5min', closed='right').sum()
Out[216]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
```

```
Freq: 5T, dtype: int64
```

你传递的频率按五分钟的增量定义了箱体边界。默认情况下，左箱体边界是包含的，因此00：00的值是包含在00：00到00：05间隔内的 [1] 。传递closed='right'将间隔的闭合端改为了右边：

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

产生的时间序列按照每个箱体左边的时间戳被标记。传递label='right'你可以使用右箱体边界标记时间序列：

```
In [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5T, dtype: int64
```

图11-3阐明了分钟频率数据按五分钟频率进行的重新采样。

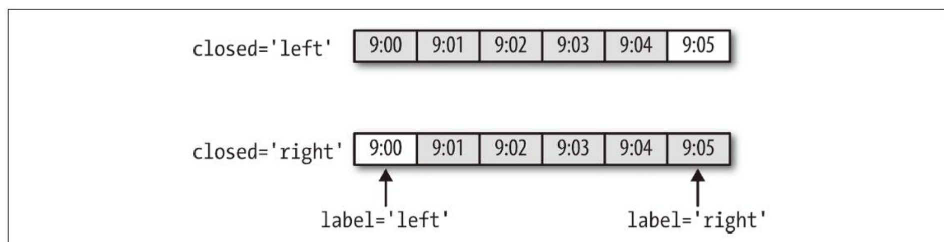


图11-3：closed、label约定的五分钟重新采样示意图

最后，你可能需要将结果索引移动一定的数量，例如从右边缘减去一秒，以使其更清楚地表明时间戳所指的间隔。要实现这个功能，向loffset传递字符串或日期偏置：

```
In [219]: ts.resample('5min', closed='right',
   .....:                 label='right', loffset='-1s').sum()
Out[219]:
1999-12-31 23:59:59     0
2000-01-01 00:04:59    15
2000-01-01 00:09:59    40
2000-01-01 00:14:59    11
Freq: 5T, dtype: int64
```

你也可以通过在结果上调用shift方法来完成loffset的效果。

11.6.1.1　开端-峰值-谷值-结束（OHLC）重新采样

在金融中，为每个数据桶计算四个值是一种流行的时间序列聚合方法：第一个值（开端）、最后一个值（结束）、最大值（峰值）和最小值（谷值）。通过使用ohlc聚合函数你将会获得包含四种聚合值列的DataFrame，这些值在数据的单次扫描中被高效计算：

```
In [220]: ts.resample('5min').ohlc()
Out[220]:
                     open  high  low  close
2000-01-01 00:00:00     0     4    0      4
2000-01-01 00:05:00     5     9    5      9
2000-01-01 00:10:00    10    11   10     11
```

[1]　closed和label的默认值选择对某些用户来说显得有些奇怪。实际上，这种选择有点武断。对于某些目标频率，closed ='left' 更可取，而其他情况下closed='right'更有意义。重要的是你要记住你是如何分隔数据的。

11.6.2 向上采样与插值

当从低频率转换为高频率时，并不需要任何聚合。让我们考虑带有每周数据的DataFrame：

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
   .....:                      index=pd.date_range('1/1/2000', perioc
   .....:                                          freq='W-WED'),
   .....:                      columns=['Colorado', 'Texas', 'New Yo:

In [222]: frame
Out[222]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

当你对这些数据使用聚合函数时，每一组只有一个值，并且会在间隙中产生缺失值。我们使用asfreq方法在不聚合的情况下转换到高频率：

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

假设你想在非星期三的日期上向前填充每周数值。fillna和reindex方法中可用的填充或插值方法可用于重采样：

```
In [225]: frame.resample('D').ffill()
Out[225]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08 -0.896431  0.677263  0.036503  0.087102
```

```
2000-01-09 -0.896431  0.677263  0.036503  0.087102
2000-01-10 -0.896431  0.677263  0.036503  0.087102
2000-01-11 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

你可以同样选择仅向前填充一定数量的区间，以限制继续使用观测值的时距：

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

请注意，新的日期索引根本不需要与旧的索引重叠：

```
In [227]: frame.resample('W-THU').ffill()
Out[227]:
            Colorado     Texas  New York      Ohio
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-13 -0.046662  0.927238  0.482284 -0.867130
```

11.6.3 使用区间进行重新采样

对以区间为索引的数据进行采样与时间戳的情况类似：

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4),
   .....:                      index=pd.period_range('1-2000', '12-20
   .....:                                            freq='M'),
   .....:                      columns=['Colorado', 'Texas', 'New Yo

In [229]: frame[:5]
Out[229]:
         Colorado     Texas  New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean()

In [231]: annual_frame
Out[231]:
      Colorado     Texas  New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

向上采样更为细致，因为你必须在重新采样前决定新频率中在时间段的哪一端放置数值，就像asfreq方法一样。convention参数默认值是'start'，但也可以是

```
'end'：
# Q-DEC: 每季度，年末在12月

In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
        Colorado     Texas  New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276

In [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
```

```
        Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

由于区间涉及时间范围，向上采样和向下采样就更为严格：

·在向下采样中，目标频率必须是原频率的子区间。

·在向上采样中，目标频率必须是原频率的父区间。

如果不满足这些规则，将会引起异常。这主要会影响每季度、每年和每周的频率。例如，根据Q-MAR定义的时间范围将只和A-MAR、A-JUN、A-SEP和A-DEC保持一致：

```
In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
        Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
2002Q1  0.046303  0.163344  0.251503 -0.157276
2002Q2  0.046303  0.163344  0.251503 -0.157276
2002Q3  0.046303  0.163344  0.251503 -0.157276
```

11.7 移动窗口函数

统计和其他通过移动窗口或指数衰减而运行的函数是用于时间序列操作的数组变换的一个重要类别。这对平滑噪声或粗糙的数据非常有用。我称这些函数为移动窗口函数，尽管它也包含了一些没有固定长度窗口的函数，比如指数加权移动平均。与其他的统计函数类似，这些函数会自动排除缺失数据。

在深入了解之前，我们可以先载入一些时间序列数据并按照工作日频率进行重新采样：

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
   .....:                            parse_dates=True, index_col=0)

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

现在我介绍rolling算子，它的行为与resample和groupby类似。rolling可以在Series或DataFrame上通过一个window（以一个区间的数字来表示，参见图11-4）进行调用。

```
In [238]: close_px.AAPL.plot()

Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f2570cf98>
In [239]: close_px.AAPL.rolling(250).mean().plot()
```

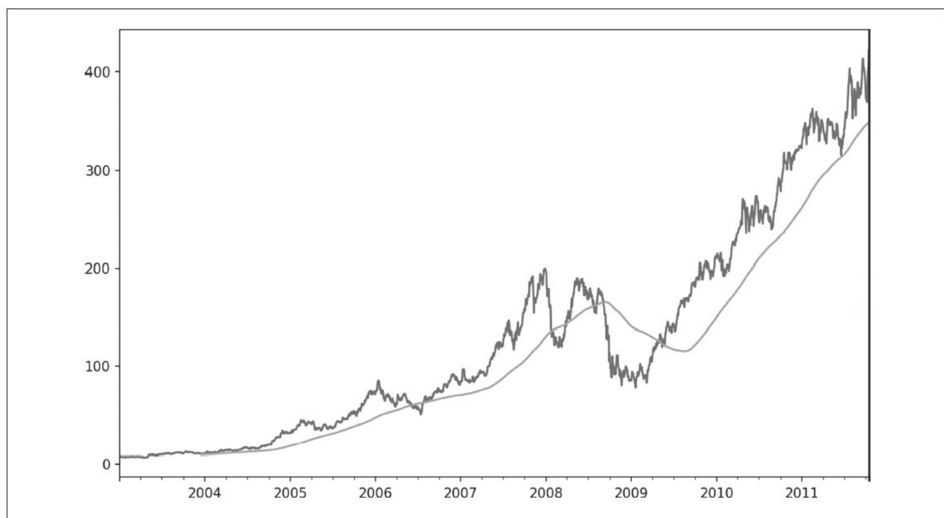


图11-4：苹果公司股价及250日MA

表达式rolling（250）与groupby的行为类似，但是它创建的对象是根据250日滑动窗口分组的而不是直接分组。因此这里我们获得了苹果公司股票价格的250日移动窗口平均值。

默认情况下，滚动函数需要窗口中所有的值必须是非NA值。由于存在缺失值这种行为会发生改变，尤其是在时间序列的起始位置你拥有的数据是少于窗口区间的（见图11-5）：

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).st

In [242]: appl_std250[5:12]
Out[242]:
2003-01-09         NaN
2003-01-10         NaN
2003-01-13         NaN
2003-01-14         NaN
2003-01-15    0.077496
2003-01-16    0.074760
2003-01-17    0.112368
Freq: B, Name: AAPL, dtype: float64

In [243]: appl_std250.plot()
```

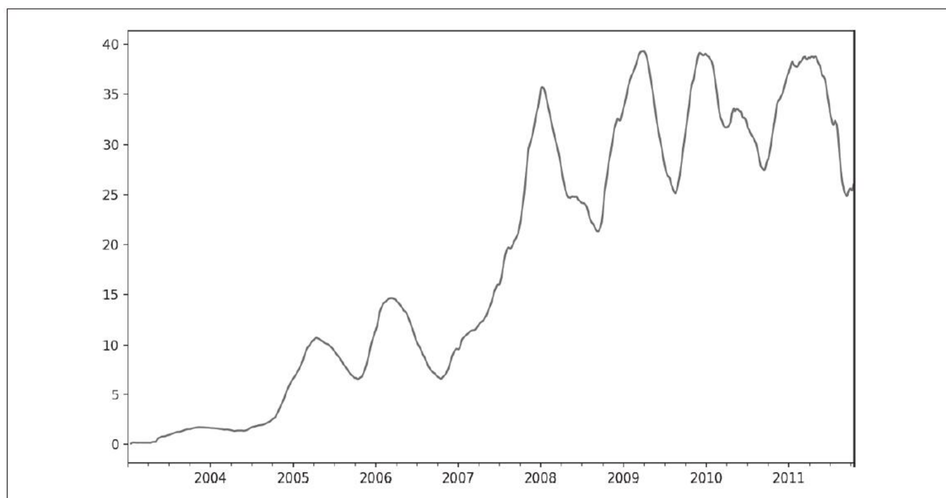


图11-5：苹果公司250日每日返回标准差

为了计算扩展窗口均值，使用expanding算子，而不是rolling。扩展均值从时间序列的起始位置开始时间窗口，并增加窗口的大小，直到它涵盖整个序列。apple_std250的扩展均值窗口如下：

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

在DataFrame上调用一个移动窗口函数会将变换应用到每一列上（见图11-6）：

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

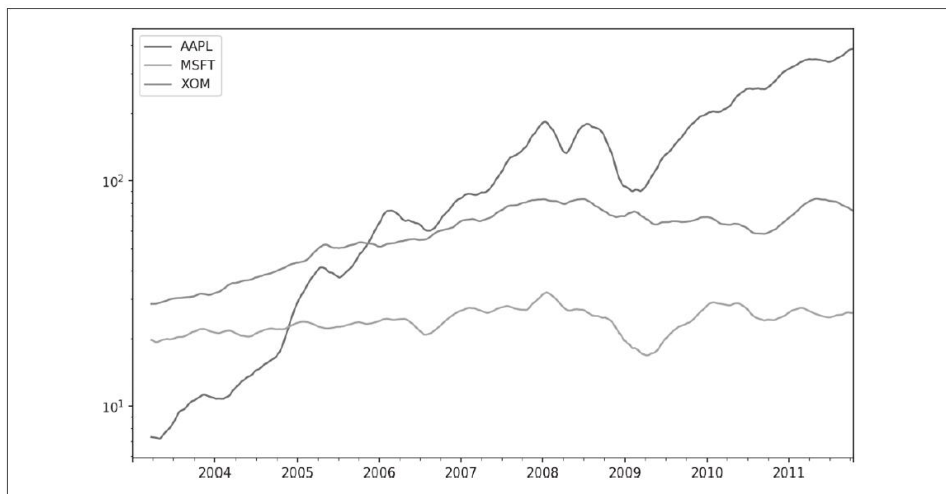


图11-6：股票价格60日MA（Y轴取对数）

rolling函数也接收表示固定大小的时间偏置字符串，而不只是一个区间的集合数字。对不规则时间序列使用注释非常有用。这些字符串可以传递给resample。例如，我们可以像这样计算20天的滚动平均值：

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
                  AAPL       MSFT        XOM
2003-01-02    7.400000  21.110000  29.220000
2003-01-03    7.425000  21.125000  29.230000
2003-01-06    7.433333  21.256667  29.473333
2003-01-07    7.432500  21.425000  29.342500
2003-01-08    7.402000  21.402000  29.240000
2003-01-09    7.391667  21.490000  29.273333
2003-01-10    7.387143  21.558571  29.238571
2003-01-13    7.378750  21.633750  29.197500
2003-01-14    7.370000  21.717778  29.194444
2003-01-15    7.355000  21.757000  29.152000
...                ...        ...        ...
2011-10-03  398.002143  25.890714  72.413571
2011-10-04  396.802143  25.807857  72.427143
2011-10-05  395.751429  25.729286  72.422857
2011-10-06  394.099286  25.673571  72.375714
2011-10-07  392.479333  25.712000  72.454667
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333
[2292 rows x 3 columns]
```

11.7.1　指数加权函数

指定一个常数衰减因子以向更多近期观测值提供更多权重，可以替代使用具有相等加权观察值的静态窗口尺寸的方法。有多种方式可以指定衰减因子。其中一种流行的方式是使用一个span（跨度），这使得结果与窗口大小等于跨度的简单移动窗口函数。

由于指数加权统计值给更近期的观测值以更多的权重，与等权重的版本相比，它对变化“适应”得更快。

pandas拥有ewm算子，同rolling、expanding算子一起使用。以下是将苹果公司股票价格的60日均线与span＝60的EW移动平均线进行比较的例子（见图11-7）：

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()
In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [254]: plt.legend()
```

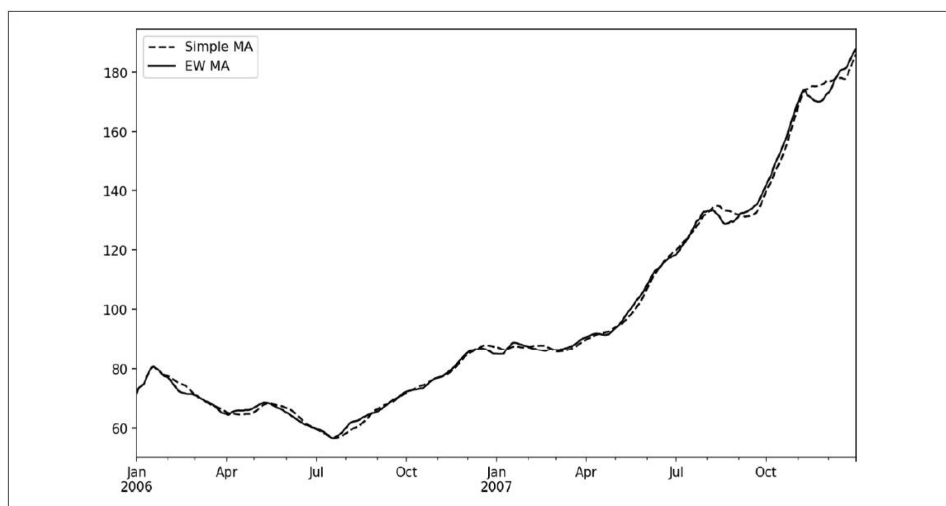


图11-7：简单移动平均和指数加权的对比

11.7.2 二元移动窗口函数

一些统计算子，例如相关度和协方差，需要操作两个时间序列。例如，金融分析师经常对股票与基准指数（如标普500）的关联性感兴趣。为了了解这个功能，我们首先计算所有我们感兴趣的时间序列的百分比变化：

```
In [256]: spx_px = close_px_all['SPX']

In [257]: spx_rets = spx_px.pct_change()

In [258]: returns = close_px.pct_change()
```

在我们调用rolling后，corr聚合函数可以根据spx_rets计算滚动相关性（见图11-8）：

```
In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_
In [260]: corr.plot()
```

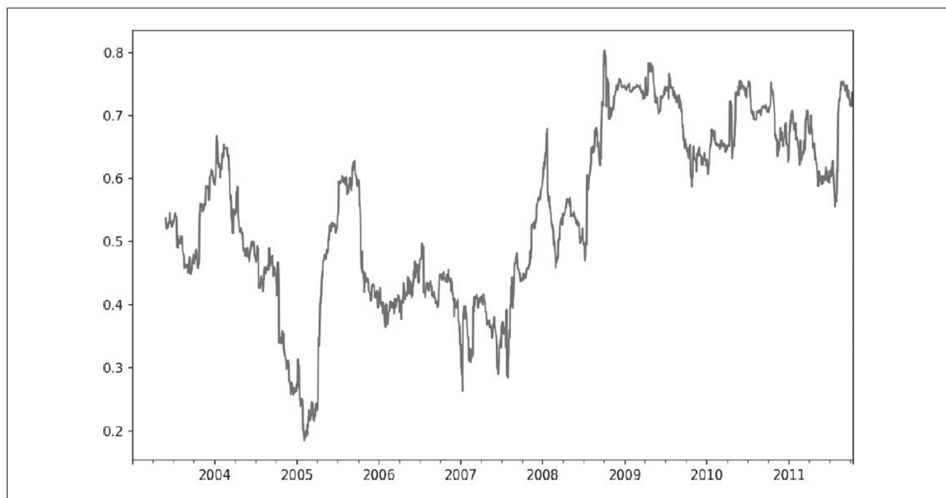


图11-8：苹果公司与标普500的六个月的收益相关性

假设你想要一次性计算多只股票与标普500的相关性。编写循环并创建一个新的DataFrame是简单的但可能也是重复性的，所以如果你传递了

一个Series或一个DataFrame，像rolling_corr这样的函数将会计算Series（例子中的spx_rets）与DataFrame中每一列的相关性（见图11-9）：

```
In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets
In [263]: corr.plot()
```

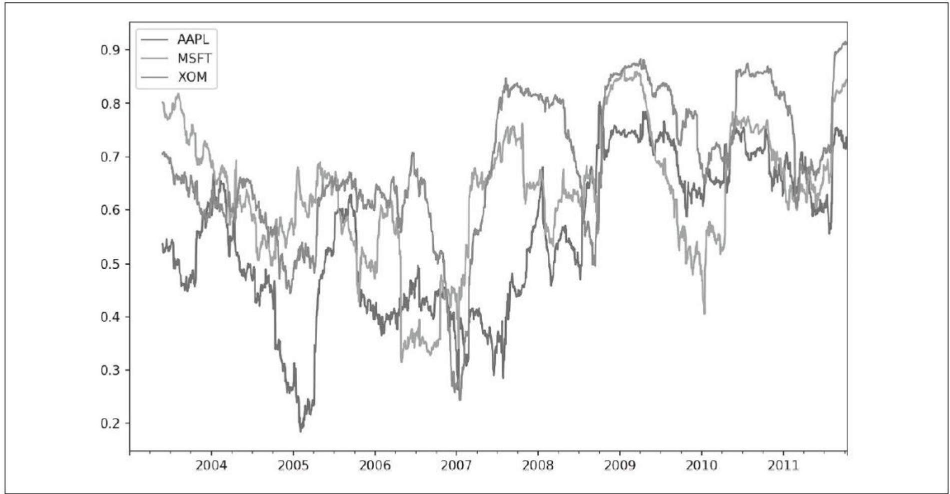


图11-9：多只股票与标普500的六个月收益相关性

11.7.3 用户自定义的移动窗口函数

在rolling及其相关方法上使用apply方法提供了一种在移动窗口中应用你自己设计的数组函数的方法。唯一的要求是该函数从每个数组中产生一个单值（缩聚）。例如，尽管我们可以使用rolling（...）.quantile（q）计算样本的分位数，但我们可能会对样本中特定值的百分位数感兴趣。scipy.stats.percentileofscore函数就是实现这个功能的（见图11-10）：

```
In [265]: from scipy.stats import percentileofscore

In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent

In [268]: result.plot()
```

如果你还没有安装SciPy，可以通过conda或pip进行安装。

11.8 本章小结

与我们在之前章节中探索的其他数据类型不同，时间序列数据需要不同的分析类型和数据转换工具。

在后续章节中，我们将会继续介绍高阶的pandas方法，并展示如何使用像statsmodels和scikit-learn这样的建模库。

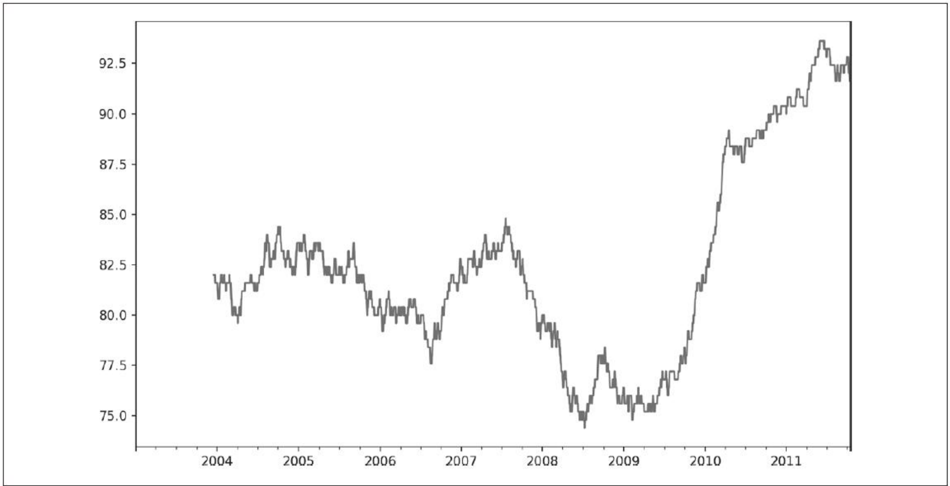


图11-10：一年窗口下苹果公司股价2%收益的百分位等级

第12章　高阶pandas

之前的章节关注于介绍不同类型的数据规则工作流以及NumPy、pandas和其他类库的特征。随着时间的推移，pandas已经开发了用于助力用户的深度特征。本章将深入一些更为高级的特征领域，帮助你深化作为一名pandas用户的专业知识。

12.1 分类数据

本节会介绍pandas的Categorical类型。我将展示在使用pandas进行某些操作时如何获得更好的性能和内存使用。我还会介绍一些在统计和机器学习应用中使用分类数据的工具。

12.1.1　背景和目标

一个列经常会包含重复值，这些重复值是一个小型的不同值的集合。我们已经看见向unique和value_counts这样的函数，它们允许我们从一个数组中提取不同值并分别计算这些不同值的频率：

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',
   ....:                     'apple'] * 2)

In [12]: values
Out[12]:
0      apple
1     orange
2      apple
3      apple
4      apple
5     orange
6      apple
7      apple
dtype: object
In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)

In [14]: pd.value_counts(values)
Out[14]:
apple     6
orange    2
dtype: int64
```

许多数据系统（用于数据入库、统计计算或其他用途）已经开发出专门的方法，用重复的值来表示数据，以便更有效地存储和计算。在数据入库的操作中，使用所谓的维度表是一种最佳实践，维度表包含了不同值，并将主要观测值存储为引用维度表的整数键：

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
Out[17]:
0     0
1     1
2     0
3     0
```

```
4    0
5    1
6    0
7    0
dtype: int64

In [18]: dim
Out[18]:
0     apple
1    orange
dtype: object
```

我们可以使用take方法来恢复原来的字符串Series：

```
In [19]: dim.take(values)
Out[19]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

这种按照整数展现的方式被称为分类或字典编码展现。不同值的数组可以被称为数据的类别、字典或层级。在本书中，我们将会使用分类的和类别这样的术语。

在你做数据分析时，分类展示会产生显著的性能提升。你也可以在类别上进行转换同时不改变代码。以下是一些相对低开销的转换示例：

·重命名类别

·在不改变已有的类别顺序的情况下添加一个新的类别

12.1.2 pandas中的Categorical类型

pandas拥有特殊的Categorical类型，用于承载基于整数的类别展示或编码的数据。让我们考虑之前的示例Series：

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,
   ....:                    'basket_id': np.arange(N),
   ....:                    'count': np.random.randint(3, 15, size=N),
   ....:                    'weight': np.random.uniform(0, 4, size=N)
   ....:                    columns=['basket_id', 'fruit', 'count', 'w

In [23]: df
Out[23]:
   basket_id   fruit  count    weight
0          0   apple      5  3.858058
1          1  orange      8  2.612708
2          2   apple      4  2.995627
3          3   apple      7  2.614279
4          4   apple     12  2.990859
5          5  orange      8  3.845227
6          6   apple      5  0.033553
7          7   apple      4  0.425778
```

这里，df['fruit']是一个Python字符串对象组成的数组。我们可以通过调用函数将它转换为Categorical对象：

```
In [24]: fruit_cat = df['fruit'].astype('category')

In [25]: fruit_cat
Out[25]:
0      apple
1     orange
2      apple
3      apple
4      apple
5     orange
6      apple
7      apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

fruit_cat的值并不是NumPy数组，而是pandas.Categorical的实例：

```
In [26]: c = fruit_cat.values

In [27]: type(c)
Out[27]: pandas.core.categorical.Categorical
```

Categorical对象拥有categories和codes属性：

```
In [28]: c.categories
Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

你可以通过分配已转换的结果将DataFrame的一列转换为Categorical对象：

```
In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit
Out[31]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

你也可以从其他Python序列类型直接生成pandas.Categorical：

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo',

In [33]: my_categories
Out[33]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

如果你已经从另一个数据源获得了分类编码数据，你可以使用from_codes构造函数：

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

除非显式地指定，分类转换是不会指定类别的顺序的。因此categories数组可能会与输入数据的顺序不同。当使用from_codes或其他任意构造函数时，你可以为类别指定一个有意义的顺序：

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,
   ....:                                         ordered=True)

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

输出的[foo < bar < baz]表明'foo'的顺序在'bar'之前，以此类推。一个未排序的分类实例可以使用as_ordered进行排序：

```
In [40]: my_cats_2.as_ordered()
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

最后需要注意，分类数据可以不是字符串，尽管我举的例子都是字符串例子。一个分类数组可以包含任一不可变的值类型。

12.1.3 使用Categorical对象进行计算

在pandas中使用Categorical与非编码版本相比（例如字符串数组）整体上是一致的。pandas中的某些部分，比如groupby函数，在与Categorical对象协同工作时性能更好。还有一些函数可以利用ordered标识。

让我们考虑一些随机数字数据，并使用pandas.qcut分箱函数。结果会返回pandas.Categorical；本书之前使用过pandas.cut，但我们略过了分类如何工作的细节：

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658])
```

让我们计算上面数据的四分位分箱，并提取一些统计值：

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.01
1, 0.63] <
                                    (0.63, 3.928]]
```

虽然样本的四分位数有用，但是在生成一份报告时，四分位数就没有四分位数名称有用了。我们可以通过在qcut函数中使用labels参数来实现这个功能：

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
```

```
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

被标记的bins分类数据并不包含数据中箱体边界的相关信息，因此我们可以使用groupby来提取一些汇总统计值：

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
   ....:            .groupby(bins)
   ....:            .agg(['count', 'min', 'max'])
   ....:            .reset_index())

In [51]: results
Out[51]:
  quartile  count       min       max
0       Q1    250 -2.949343 -0.685484
1       Q2    250 -0.683066 -0.010115
2       Q3    250 -0.010032  0.628894
3       Q4    250  0.634238  3.927528
```

结果中的'quartile'列保留了bins中原始的分类信息，包括顺序：

```
In [52]: results['quartile']
Out[52]:
0    Q1
1    Q2
2    Q3
3    Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

12.1.3.1 使用分类获得更高性能

如果你在特定的数据集上做了大量的分析，将数据转换为分类数据可以产生大幅的性能提升。DateFrame中一列的分类版本通常也会明显使用更少内存。让我们考虑一些含有一千万元素的Series以及少量的不同类别：

```
In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))
```

```
In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

现在我们将labels转换为Categorical对象：

```
In [56]: categories = labels.astype('category')
```

现在我们注意到labels比categories使用了明显更多的内存：

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

分类转换当然不是免费的，但是它是一次性开销：

```
In [59]: %time _ = labels.astype('category')
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

使用分类对象进行GroupBy操作明显更快，这是因为底层算法使用了基于整数代码的数组而不是字符串数组。

12.1.4　分类方法

Series包含的分类数据拥有一些特殊方法，这些方法类似于Series.str的特殊字符串方法。这些方法提供了快捷访问类别和代码的方式。考虑下面的Series：

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

特殊属性cat提供了对分类方法的访问：

```
In [63]: cat_s.cat.codes
Out[63]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

假设我们知道该数据的实际类别集合超出了数据中观察到的四个值。我们可以使用set_categories方法来改变类别：

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [67]: cat_s2
Out[67]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

虽然看起来数据并未改变，但新类别将反映在使用它们的操作中。例如，value_counts将遵循新的类别（如果存在）：

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64

In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

在大型数据集中，分类数据经常被用于节省内存和更高性能的便捷工具。在你过滤了一个大型DataFrame或Series之后，很多类别将不会出现在数据中。为了帮助解决这个问题，我们可以使用remove_unused_categories方法来去除未观察到的类别：

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [71]: cat_s3
Out[71]:
0    a
```

```
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]
```

参见表12-1中可用的分类方法列表。

表12-1：pandas中Series的分类方法

方法	描述
add_categories	将新的类别（未使用过的）添加到已有类别的尾部
as_ordered	对类别排序
as_unordered	使类别无序
remove_categories	去除类别，将被移除的值置为 null
remove_unused_categories	去除所有没有出现在数据中的类别
rename_categories	使用新的类别名称替代现有的类别，不会改变类别的数量
reorder_categories	与 rename_categories 类似，但结果是经过排序的类别
set_categories	用指定的一组新类别替换现有类别，可以添加或删除类别

12.1.4.1 创建用于建模的虚拟变量

当你使用统计数据或机器学习工具时，通常会将分类数据转换为虚拟变量，也称为one-hot编码。这会产生一个DataFrame，每个不同的类别都是它的一列。这些列包含一个特定类别的出现次数，否则为0。

考虑之前的一个例子：

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category
```

之前在第7章中提到过，pandas.get_dummies函数将一维的分类数据转换为一个包含虚拟变量的DataFrame：

```
In [74]: pd.get_dummies(cat_s)
Out[74]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

12.2　高阶GroupBy应用

虽然我们已经在第10章深入讨论了在Series和DataFrame使用groupby方法，但仍然有一些额外的方法你可能会用到。

12.2.1 分组转换和“展开”GroupBy

在第10章中，我们在分组操作中学习了apply方法用于执行转换操作。还有另一个内建方法transform，与apply方法类似但是会对你可以使用的函数种类加上更多的限制：

·transform可以产生一个标量值，并广播到各分组的尺寸数据中

·transform可以产生一个与输入分组尺寸相同的对象

·transform不可改变它的输入

让我们考虑一个简单的例子：

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
   ....:                    'value': np.arange(12.)})

In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

这里是按'key'分组的均值：

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

假设我们想要产生一个Series，它的尺寸和df['value']一样，但值都被按'key'分组的均值替代。我们可以向transfrom传递匿名函数lambda x：x.mean（）：

```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

对于内建的聚合函数，我们可以像GroupBy的agg方法一样传递一个字符串别名：

```
In [80]: g.transform('mean')
Out[80]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

与apply类似，transform可以与返回Series的函数一起使用，但结果必须和输入有相同的大小。例如，我们可以使用lambda函数给每个组乘以2：

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
```

```
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64
```

作为一个更复杂的例子，我们可以按照每个组的降序计算排名：

```
In [82]: g.transform(lambda x: x.rank(ascending=False))
Out[82]:
0     4.0
1     4.0
2     4.0
3     3.0
4     3.0
5     3.0
6     2.0
7     2.0
8     2.0
9     1.0
10    1.0
11    1.0
Name: value, dtype: float64
```

考虑一个由简单聚合构成的分组转换函数：

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

我们使用transform或apply可以获得等价的结果：

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
```

```
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

内建的聚合函数如'mean'或'sum'通常会比apply函数更快。这些函数在与transform一起使用时也会存在一个"快速通过"。这允许我们执行一个所谓的展开分组操作：

```
In [86]: g.transform('mean')
Out[86]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64

In [87]: normalized = (df['value'] - g.transform('mean')) / g.transf

In [88]: normalized
Out[88]:
0    -1.161895
1    -1.161895
```

```
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

然而一个展开分组操作可能会包含多个分组聚合，矢量化操作的整体优势往往超过了这一点。

12.2.2 分组的时间重新采样

对于时间序列数据，resample方法在语义上是一种基于时间分段的分组操作。下面是一个小的示例表：

```
In [89]: N = 15

In [90]: times = pd.date_range('2017-05-20 00:00', freq='1min', peri

In [91]: df = pd.DataFrame({'time': times,
    ....:                    'value': np.arange(N)})

In [92]: df
Out[92]:
                  time  value
0  2017-05-20 00:00:00      0
1  2017-05-20 00:01:00      1
2  2017-05-20 00:02:00      2
3  2017-05-20 00:03:00      3
4  2017-05-20 00:04:00      4
5  2017-05-20 00:05:00      5
6  2017-05-20 00:06:00      6
7  2017-05-20 00:07:00      7
8  2017-05-20 00:08:00      8
9  2017-05-20 00:09:00      9
10 2017-05-20 00:10:00     10
11 2017-05-20 00:11:00     11
12 2017-05-20 00:12:00     12
13 2017-05-20 00:13:00     13
14 2017-05-20 00:14:00     14
```

这里，我们可以按'time'进行索引，然后重新采样：

```
In [93]: df.set_index('time').resample('5min').count()
Out[93]:
                     value
time
2017-05-20 00:00:00      5
2017-05-20 00:05:00      5
2017-05-20 00:10:00      5
```

假设DataFrame包含多个时间序列，并按一个附加的分组键列进行了标记：

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
   ....:                     'key': np.tile(['a', 'b', 'c'], N),
   ....:                     'value': np.arange(N * 3.)})

In [95]: df2[:7]
Out[95]:
  key                time  value
0   a 2017-05-20 00:00:00    0.0
1   b 2017-05-20 00:00:00    1.0
2   c 2017-05-20 00:00:00    2.0
3   a 2017-05-20 00:01:00    3.0
4   b 2017-05-20 00:01:00    4.0
5   c 2017-05-20 00:01:00    5.0
6   a 2017-05-20 00:02:00    6.0
```

要为每个'key'的值进行相同的重新采样，我们可以使用pandas.TimeGrouper对象：

```
In [96]: time_key = pd.TimeGrouper('5min')
```

之后我们可以设置时间索引，按'key'和time_key进行分组，再聚合：

```
In [97]: resampled = (df2.set_index('time')
   ....:              .groupby(['key', time_key])
   ....:              .sum())

In [98]: resampled
Out[98]:
                         value
key time
a   2017-05-20 00:00:00   30.0
    2017-05-20 00:05:00  105.0
    2017-05-20 00:10:00  180.0
b   2017-05-20 00:00:00   35.0
    2017-05-20 00:05:00  110.0
    2017-05-20 00:10:00  185.0
c   2017-05-20 00:00:00   40.0
    2017-05-20 00:05:00  115.0
    2017-05-20 00:10:00  190.0

In [99]: resampled.reset_index()
Out[99]:
  key                time  value
0   a 2017-05-20 00:00:00   30.0
1   a 2017-05-20 00:05:00  105.0
2   a 2017-05-20 00:10:00  180.0
3   b 2017-05-20 00:00:00   35.0
4   b 2017-05-20 00:05:00  110.0
5   b 2017-05-20 00:10:00  185.0
```

```
6    c 2017-05-20 00:00:00   40.0
7    c 2017-05-20 00:05:00  115.0
8    c 2017-05-20 00:10:00  190.0
```

使用TimeGrouper的一个限制是时间必须是Series或DataFrame的索引。

12.3 方法链技术

在向数据集应用一系列变换时，你可能会发现自己创建了许多临时变量，而这些变量在分析中从未使用过。例如，考虑以下例子：

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean()
result = df2.groupby('key').col1_demeaned.std()
```

尽管我们在这里并未使用真实数据，但是这个例子体现了一些新的方法。首先，DataFrame.assign方法是对df[k] = v的赋值方式的一种功能替代。它返回的是一个按指定修改的新的DataFrame，而不是在原对象上进行修改。因此，下面这些表述是等价的：

```
# 常见的非函数方式
df2 = df.copy()
df2['k'] = v

# 函数赋值的方式
df2 = df.assign(k=v)
```

原位赋值可能比使用assign更为快速，但assign可以实现更方便的方法链：

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2.mean())
          .groupby('key')
          .col1_demeaned.std())
```

我使用外部的小括号来使添加换行符更方便。

在做方法链时要牢记你可能会需要引用临时对象。在之前的例子中，我们无法引用load_data的结果，除非它被赋值给临时变量df。为了处理这种情况，assign和很多其他的pandas函数接受函数型的参数，这种参数也被称为可调用参数。

为了展示操作中的可调用对象，考虑下面这段之前讲过的代码段：

```
df = load_data()
df2 = df[df['col2'] < 0]
```

上面的代码可以改写为：

```
df = (load_data()
      [lambda x: x['col2'] < 0])
```

这里，load_data的结果没有复制给一个变量，因此传递进[]的函数将会被绑定到方法链那一阶段的对象上。

之后，我们可以继续将整个序列写作一个单链表达式：

```
result = (load_data()
          [lambda x: x.col2 < 0]
          .assign(col1_demeaned=lambda x: x.col1 - x.col1.mean())
          .groupby('key')
          .col1_demeaned.std())
```

无论你是否倾向于按这种风格写代码都是偏好问题，但将表达式分解为多个步骤可能会使代码更具可读性。

12.3.1 pipe方法

使用内建的pandas函数和我们刚才看到的用可调用参数进行方法链接的方式，你可以完成很多工作。然而，有时你需要使用自定义的函数或来自第三方库的函数。这就是pipe（管道）方法出现的原因。

考虑下面一个函数调用序列：

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```

在使用接受并返回Series或DataFrame对象的函数时，你可以调用pipe方法重写代码：

```
result = (df.pipe(f, arg1=v1)
          .pipe(g, v2, arg3=v3)
          .pipe(h, arg4=v4))
```

表达式f（df）和df.pipe（f）是等价的，但是pipe使得链式调用更为方便。

将操作的序列泛化成可复用的函数是pipe方法的一个潜在用途。作为示例，让我们考虑从一列中减去分组平均值：

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

假设你想要对多个列去除均值并方便地改变分组键。此外，你可能想要将转换在方法链中执行。下面是一个示例实现：

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
    return result
```

之后可以写如下代码：

```
result = (df[df.col1 < 0]
          .pipe(group_demean, ['key1', 'key2'], ['col1']))
```

12.4 本章小结

和很多其他的开源软件项目一样，pandas正在持续变化并获得新的、升级的功能。与本书的其他章节一样，我们主要关注的是大多数稳定功能，这些功能在未来的一些年中都不太会发生变化。

为了深化你作为pandas用户的专业水平，我推荐你探索官方文档（http://pandas.pydata.org ）并阅读发布纪要，发布纪要记录了开发团队新的开源发布内容。我们也邀请你加入pandas的开发：修复漏洞、建立新特性、提升文档水平等。

第13章　Python建模库介绍

在本书中，我专注于提供一个用Python做数据分析的编程基础。因为数据分析师和科学家经常在数据规整和准备上花费大量的时间，本书的结构反映了精通这些技术的重要性。

使用哪个库进行模型开发取决于应用。很多统计上的难题可以通过更简单的技术，例如最小二乘回归等方法，来解决，但另外一些问题可能需要用到高阶的机器学习方法。幸运的是Python已经成为实现分析方法的语言选择之一，因此你在完成本书学习后可以探索很多工具的使用。

在本章，我将回顾pandas的一些特性，这些特性可能会在你使用pandas进行模型训练和评分时有用。我将介绍两个流行的建模工具包——statsmodels（http://statsmodels.org ）和scikit-learn（http://scikit-learn.org ）。由于这两个项目每个都大到可以单独成书，我就不再尝试完整地介绍，而是像其他基于Python的数据科学、统计学和机器学习的书籍一样，直接带你进入到项目的官方在线文档。

13.1 pandas与建模代码的结合

使用pandas用于数据载入和数据清洗，之后切换到模型库去建立模型是一个常见的模型开发工作流。在机器学习中，特征工程是模型开发的重要部分之一。特征工程是指从原生数据集中提取可用于模型上下文的有效信息的数据转换过程或分析。

尽管一个“好”的特征工程的细节已经超出了本书的范围，但我仍然会展示一些可以在利用pandas进行数据操作和建模之间无痛切换的方法。

pandas和其他分析库的结合点通常是NumPy数组。要将DataFrame转换为NumPy数组，使用.values属性：

```
In [10]: import pandas as pd
In [11]: import numpy as np

In [12]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1.  ,  0.01, -1.5 ],
       [ 2.  , -0.01,  0.  ],
       [ 3.  ,  0.25,  3.6 ],
       [ 4.  , -4.1 ,  1.3 ],
       [ 5.  ,  0.  , -2.  ]])
```

将数组再转换为DataFrame，你可能会回忆起之前的章节，你可以传递一个含有列名的二维ndarray：

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'thre

In [17]: df2
Out[17]:
   one   two  three
0  1.0  0.01   -1.5
1  2.0 -0.01    0.0
2  3.0  0.25    3.6
3  4.0 -4.10    1.3
4  5.0  0.00   -2.0
```

.values属性一般在你的数据是同构化的时候使用——例如，都是数字类型的时候。如果你的数据是异构化的，结果将是Python对象的ndarray：

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
   x0    x1    y strings
0   1  0.01 -1.5       a
1   2 -0.01  0.0       b
2   3  0.25  3.6       c
3   4 -4.10  1.3       d
4   5  0.00 -2.0       e

In [21]: df3.values
Out[21]:
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

对于某些模型，你可能只想使用一部分列。我推荐使用loc索引和values：

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1.  ,  0.01],
       [ 2.  , -0.01],
       [ 3.  ,  0.25],
       [ 4.  , -4.1 ],
```

```
         [ 5.  ,  0.  ]])
```

有些库对pandas有本地化支持，可以自动为你做以下工作：将数据从DataFrame转换到NumPy中并将模型参数名称附于输出表的列或Series上。在其他情况下，你将不得不手动去处理这些“元数据管理”的操作。

在第12章，我们学习了pandas的Categorical类型和pandas.get_dummies函数。假设在我们的示例数据集中，我们有一个非数字类型的列：

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
   ....:                                   categories=['a', 'b'])

In [25]: data
Out[25]:
   x0    x1    y category
0   1  0.01 -1.5        a
1   2 -0.01  0.0        b
2   3  0.25  3.6        a
3   4 -4.10  1.3        a
4   5  0.00 -2.0        b
```

如果我们想使用虚拟变量替代'category'列，我们先创建虚拟变量，之后删除'categroy'列，然后连接结果：

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')
In [27]: data_with_dummies = data.drop('category', axis=1).join(dumm

In [28]: data_with_dummies
Out[28]:
   x0    x1    y  category_a  category_b
0   1  0.01 -1.5           1           0
1   2 -0.01  0.0           0           1
2   3  0.25  3.6           1           0
3   4 -4.10  1.3           1           0
4   5  0.00 -2.0           0           1
```

在使用虚拟变量拟合特定的统计模型时是有一些细微区别的。当你拥有不止简单的数字类型列时，使用Patsy（下一节的内容）可以更简单、更少出错。

13.2 使用Patsy创建模型描述

Patsy（https://patsy.readthedocs.io/ ）是一个用于描述统计模型（尤其是线性模型）的Python库。它使用一种小型基于字符串的“公式语法”，这种语法受到了R、S统计编程语言中公式语法的启发。

Patsy能够很好地支持statsmodels中特定的线性模型，因此我将专注于它的主要特性，帮助你把程序跑起来。Patsy的公式是特殊字符串语法，如下：

```
y ~ x0 + x1
```

语法a＋b并不代表a加b，而是指为模型而创建的设计矩阵中的名词列。patsy.dmatrices函数在数据集上（可以是一个DataFrame或数组的字典）使用了一个公式字符串，并为一个线性模型产生了设计矩阵：

```
In [29]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

现在我们可以得到：

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
```

```
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

这些Patsy的DesignMatrix实例是含有附加元数据的NumPy ndarray：

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1.  ,  1.  ,  0.01],
       [ 1.  ,  2.  , -0.01],
       [ 1.  ,  3.  ,  0.25],
       [ 1.  ,  4.  , -4.1 ],
       [ 1.  ,  5.  ,  0.  ]])
```

你们可能会猜想Intercept（截距）这个名词列的由来。这其实是线性模型，如最小二乘回归（OLS，Ordinary Least Squares）中的惯例。你可以通过给模型添加名词列+0来加入截距：

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
```

```
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

Patsy对象可以直接传递给一些算法，比如numpy.linalg.lstsq等，这些算法都会执行一个最小二乘回归：

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

模型元数据保留在design_info属性中，因此你可以将模型列名重新附加到拟合系数以获得一个Series，例如：

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_

In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

13.2.1 Patsy公式中的数据转换

你可以将Python代码混合到你的Patsy公式中，在执行公式时，Patsy库将尝试在封闭作用域中寻找你使用的函数：

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', da

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

一些常用的变量转换包括标准化（对均值0和方差1）和居中（减去平均值）。为了实现这个目的，Patsy具有内置函数：

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', d

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
  Intercept  standardize(x0)  center(x1)
          1         -1.41421        0.78
          1         -0.70711        0.76
          1          0.00000        1.02
          1          0.70711       -3.33
          1          1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

作为建模的一部分过程，你可能会在一个数据集上拟合一个模型，之后基于另一个模型评价该模型。这个过程可以保留部分数据或者之后再加入新数据。在应用像居中和标准化这样的转换时，你在基于新数据使用模型进行预测的时候要小心。这些转换被称为有状态的转换，因为你在形成

新数据集时必须使用原数据集中的均值或标准差等统计值。

patsy.build_design_matrices函数可以使用原始样本内数据集中保存的信息将变换应用于新的样本外数据上：

```
In [46]: new_data = pd.DataFrame({
   ....:     'x0': [6, 7, 8, 9],
   ....:     'x1': [3.1, -0.5, 0, 2.3],
   ....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_dat

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
   Intercept  standardize(x0)  center(x1)
           1          2.12132        3.87
           1          2.82843        0.27
           1          3.53553        0.77
           1          4.24264        3.07
   Terms:
     'Intercept' (column 0)
     'standardize(x0)' (column 1)
     'center(x1)' (column 2)]
```

因为加号（+）在Patsy公式的上下文中并不是加法的意思，当你想要对数据集中两列按列名相加时，你必须将列名封装到特殊的I函数中：

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
  Intercept  I(x0 + x1)
          1        1.01
          1        1.99
          1        3.25
          1       -0.10
          1        5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

在patsy.builtins模块中Patsy还有其他一些内建转换。更多内容请参看官方在线文档。

分类数据有一类特殊的转换，我将在之后介绍。

13.2.2 分类数据与Patsy

有多种方式可以将非数字类型数据转换以用于模型的设计矩阵。一个完整的解决方案是超出了本书的范围的，最好是能够学习一门统计学课程。

当你在Patsy公式中使用非数字名词列时，它们将会被默认转换为虚拟变量。如果有拦截，其中一个级别将被排除以避免共线性：

```
In [51]: data = pd.DataFrame({
   ....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
   ....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
   ....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
   ....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
   ....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
  Intercept  key1[T.b]
          1          0
          1          0
          1          1
          1          1
          1          0
          1          1
          1          0
          1          1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
```

如果你忽略了模型的截距，每个类别值的列将会被包含在模型的设计矩阵中：

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
  key1[a]  key1[b]
        1        0
        1        0
        0        1
```

```
        0        1
        1        0
        0        1
        1        0
        0        1
  Terms:
    'key1' (columns 0:2)
```

数字类型列可以使用C函数解释为分类类型：

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept  C(key2)[T.1]
          1             0
          1             1
          1             0
          1             1
          1             0
          1             1
          1             0
          1             0
  Terms:
    'Intercept' (column 0)
    'C(key2)' (column 1)
```

当你在模型中使用多个分类名词列时，事情可能会更加复杂，因为你可以包含形式为key1：key2的交互项，例如，可用于方差分析（ANOVA）模型：

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})

In [59]: data
Out[59]:
  key1  key2  v1   v2
0    a  zero   1 -1.0
1    a   one   2  0.0
2    b  zero   3  2.5
3    b   one   4 -0.5
4    a  zero   5  4.0
5    b   one   6 -1.2
6    a  zero   7  0.2
7    b  zero   8 -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)
```

```
In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept  key1[T.b]  key2[T.zero]
          1          0             1
          1          0             0
          1          1             1
          1          1             0
          1          0             1
          1          1             0
          1          0             1
          1          1             1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)

In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
  Intercept  key1[T.b]  key2[T.zero]  key1[T.b]:key2[T.zero]
          1          0             1                       0
          1          0             0                       0
          1          1             1                       1
          1          1             0                       0
          1          0             1                       0
          1          1             0                       0
          1          0             1                       0
          1          1             1                       1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)
    'key1:key2' (column 3)
```

Patsy提供了其他几种方式，可以转换分类数据，包括按照特定顺序对名词列进行转换。更多内容请参看官方在线文档。

13.3 statsmodels介绍

statsmodels（http://www.statsmodels.org ）是一个Python库，用于拟合多种统计模型，执行统计测试以及数据探索和可视化。statsmodels包含更多的“经典”频率学派统计方法，而贝叶斯方法和机器学习模型可在其他库中找到。

包含在statsmodels中的一些模型：

·线性模型，广义线性模型和鲁棒线性模型

·线性混合效应模型

·方差分析（ANOVA）方法

·时间序列过程和状态空间模型

·广义的矩量法

在接下来的几页中，我们将在statsmodels中使用一些基本工具，并探讨如何使用带有Patsy公式和pandas DataFrame对象的建模接口。

13.3.1 评估线性模型

统计模型中有几种线性回归模型，从较基本的（例如，普通最小二乘）到更复杂的（例如，迭代重新加权的最小二乘）。

statsmodels中的线性模型有两个不同的主要接口：基于数组的和基于公式的。这些接口通过这些API模块导入来访问：

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

为了展示如何使用这些，我将根据一些随机数据生成线性模型：

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)

# 用于复现
np.random.seed(12345)
N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

在这里，我写下了已知参数beta的“真实”模型。在这种情况下，dnorm是用于生成具有特定均值和方差的正态分布数据的辅助函数。所以现在我们有：

```
In [66]: X[:5]
Out[66]:
array([[-0.1295, -1.2128,  0.5042],
       [ 0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253,  0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])

In [67]: y[:5]
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895, -0.1289])
```

线性模型通常与我们在Patsy中看到的截距项相匹配。sm.add_constant函数可以将截距列添加到现有矩阵：

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1.    , -0.1295, -1.2128,  0.5042],
       [ 1.    ,  0.3029, -0.4357, -0.2542],
       [ 1.    , -0.3285, -0.0253,  0.1384],
       [ 1.    , -0.3515, -0.7196, -0.2582],
       [ 1.    ,  1.2433, -0.3738, -0.5226]])
```

sm.OLS类可以拟合一个最小二乘线性回归：

```
In [70]: model = sm.OLS(y, X)
```

模型的fit方法返回一个回归结果对象，该对象包含了估计的模型参数和其他的诊断：

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([ 0.1783,  0.223 ,  0.501 ])
```

在results上调用summary方法可以打印出一个模型的诊断细节：

```
In [73]: print(results.summary())
OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 25 Sep 2017   Prob (F-statistic):
Time:                        14:06:15   Log-Likelihood:
No. Observations:                 100   AIC:
Df Residuals:                      97   BIC:
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
------------------------------------------------------------------------------
```

```
x1             0.1783      0.053      3.364      0.001      0.073
x2             0.2230      0.046      4.818      0.000      0.131
x3             0.5010      0.080      6.237      0.000      0.342
==============================================================================
Omnibus:                        4.662   Durbin-Watson:
Prob(Omnibus):                  0.097   Jarque-Bera (JB):
Skew:                           0.481   Prob(JB):
Kurtosis:                       3.243   Cond. No.
==============================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors i
specified.
```

此处的参数名称已被赋予通用名称x1、x2等。假设所有模型参数都在DataFrame中：

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
       col0      col1      col2         y
0 -0.129468 -1.212753  0.504225  0.427863
1  0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302  0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4  1.243269 -0.373799 -0.522629 -0.128941
```

现在我们可以使用statsmodels公式API和Patsy公式字符串：

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit(

In [78]: results.params
Out[78]:
Intercept    0.033559
col0         0.176149
col1         0.224826
col2         0.514808
dtype: float64

In [79]: results.tvalues
Out[79]:
Intercept    0.952188
col0         3.319754
col1         4.850730
col2         6.303971
dtype: float64
```

观察statsmodels如何将结果作为带有DataFrame列名称的Series返回。使用公式和pandas对象时，我们也不需要使用add_constant。

给定新的样本外数据后，你可以根据估计的模型参数计算预测值：

```
In [80]: results.predict(data[:5])
Out[80]:
0   -0.002327
1   -0.141904
2    0.041226
3   -0.323070
4   -0.100535
dtype: float64
```

还有很多额外的工具，可针对你在statsmodels中能够探索的线性模型结果进行分析、诊断和可视化。除了普通最小二乘法之外，还有其他种类的线性模型。

13.3.2 评估时间序列处理

statsmodels中的另一类模型用于时间序列分析。其中包括自回归过程，卡尔曼滤波和其他状态空间模型，以及多变量自回归模型。

让我们模拟一些具有自回归结构和噪声的时间序列数据：

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

该数据具有参数为0.8和-0.4的AR（2）结构（两个滞后）。当你拟合一个AR模型时，你可能不知道包含的滞后项的数量，所以你可以用更大的滞后数来拟合该模型：

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

结果中的估计参数首先是截距，接下来是前两个滞后的估计：

```
In [85]: results.params
Out[85]: array([-0.0062,  0.7845, -0.4085, -0.0136,  0.015 ,  0.0143
```

这些模型的深层细节以及如何解释其结果超出了本书所能涵盖的范围，但在statsmodels文档中还有很多可以发现的内容。

13.4 scikit-learn介绍

scikit-learn（http://scikit-learn.org ）是使用最广泛且最受信任的通用Python机器学习库。它包含广泛的标准监督的和无监督的机器学习方法，包括用于模型选择和评估、数据转换、数据加载和模型持久化的工具。这些模型可用于分类、聚类、预测和其他常见任务。

有很多优秀的在线和印刷资源可用于学习机器学习，以及如何应用scikit-learn和TensorFlow等库来解决实际问题。在本节中，我将简要介绍一下scikit-learn API风格。

在写这篇文章的时候，scikit-learn并没有深度地和pandas集成，尽管还有一些附加的第三方包仍在开发中。不过，在模型拟合之前，pandas对于“按摩”数据集非常有用。

作为一个例子，我使用Kaggle比赛（https://www.kaggle.com/c/titanic ）中关于泰坦尼克号上生还乘客的经典数据集，其中泰坦尼克号于1912年沉没。我们使用pandas载入测试和训练数据集：

```
In [86]: train = pd.read_csv('datasets/titanic/train.csv')

In [87]: test = pd.read_csv('datasets/titanic/test.csv')

In [88]: train[:4]
Out[88]:
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
                                                Name     Sex   Age  Si
0                            Braund, Mr. Owen Harris    male  22.0
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
2                             Heikkinen, Miss. Laina  female  26.0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
```

像statsmodels和scikit-learn通常不能提供缺失数据，因此我们要检查各列，看看是否有包含缺失数据：

```
In [89]: train.isnull().sum()
Out[89]:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64

In [90]: test.isnull().sum()
Out[90]:
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

在像这样的统计和机器学习的例子中，一个典型的任务是根据数据中的特征来预测乘客是否能幸存下来。将模型拟合到训练数据集上，然后在样本外测试数据集上进行评估。

我想用Age作为预测，但它缺少数据。有很多方法可以进行缺失数据插补（imputation），但我会做一个简单的插补，并使用训练数据集的中间值填充两个表中的空值：

```
In [91]: impute_value = train['Age'].median()

In [92]: train['Age'] = train['Age'].fillna(impute_value)

In [93]: test['Age'] = test['Age'].fillna(impute_value)
```

现在我们需要明确我们的模型。我添加了一列IsFemale作为'Sex'列的编码版本：

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [95]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

然后我们决定一些模型变量并创建NumPy数组：

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']

In [97]: X_train = train[predictors].values

In [98]: X_test = test[predictors].values

In [99]: y_train = train['Survived'].values

In [100]: X_train[:5]
Out[100]:
array([[  3.,   0.,  22.],
       [  1.,   1.,  38.],
       [  3.,   1.,  26.],
       [  1.,   1.,  35.],
       [  3.,   0.,  35.]])

In [101]: y_train[:5]
Out[101]: array([0, 1, 1, 1, 0])
```

我没有声称这是一个好的模型，也没有说这些功能是正确设计的。我们使用scikit-learn的LogisticRegression模型创建一个模型实例：

```
In [102]: from sklearn.linear_model import LogisticRegression

In [103]: model = LogisticRegression()
```

与statsmodels类似，我们可以使用模型的fit方法在训练数据上拟合模型：

```
In [104]: model.fit(X_train, y_train)
Out[104]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercep
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jol
          penalty='l2', random_state=None, solver='liblinear', tol=0
          verbose=0, warm_start=False)
```

现在，我们可以使用model.predict为测试数据集形成预测：

```
In [105]: y_predict = model.predict(X_test)

In [106]: y_predict[:10]
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

如果你拥有测试数据集的真实值，则可以计算精度百分比或其他一些错误指标：

```
(y_true == y_predict).mean()
```

实际上，模型训练中经常存在许多附加的复杂层次。许多模型具有可以调整的参数，并且存在可用于参数调整的交叉验证等技术以避免过度拟合训练数据。这通常可以在新数据上产生更好的预测性能或稳健性。

交叉验证通过分割训练数据来模拟样本外预测。基于像均方误差之类的模型准确度分数，可以对模型参数执行网格搜索。一些模型，如逻辑回归，具有内置交叉验证的估计类。例如，LogisticRegressionCV类可以与一个参数一起使用，该参数表示网格搜索在模型正则化参数C上的细致度：

```
In [107]: from sklearn.linear_model import LogisticRegressionCV

In [108]: model_cv = LogisticRegressionCV(10)

In [109]: model_cv.fit(X_train, y_train)
Out[109]:
LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
           fit_intercept=True, intercept_scaling=1.0, max_iter=100,
           multi_class='ovr', n_jobs=1, penalty='l2', random_state=No
           refit=True, scoring=None, solver='lbfgs', tol=0.0001, verl
```

要手动进行交叉验证，可以使用cross_val_score帮助函数，该函数处理数据拆分过程。例如，为了用我们的模型与训练数据的四个非重叠分割进行交叉验证，我们可以这样做：

```
In [110]: from sklearn.model_selection import cross_val_score

In [111]: model = LogisticRegression(C=10)

In [112]: scores = cross_val_score(model, X_train, y_train, cv=4)
```

```
In [113]: scores
Out[113]: array([ 0.7723,  0.8027,  0.7703,  0.7883])
```

默认评分指标是依赖于模型的，但可以选择明确的评分函数。经过交叉验证的模型需要更长时间的训练，但通常可以产生更好的模型性能。

13.5 继续你的教育

虽然我只带读者浏览了一些Python建模库的皮毛，但是有越来越多的框架用于各种统计和机器学习，可以用Python或用户界面来实现。

本书特别关注数据规整，但还有许多其他书籍包含了关于建模和数据科学工具的内容。这些优秀的书籍包括：

·Introduction to Machine Learning with Python by Andreas Mueller and Sarah Guido（O’Reilly）

·Python Data Science Handbook by Jake VanderPlas（O’Reilly）

·Data Science from Scratch: First Principles with Python by Joel Grus（O’Reilly）

·Python Machine Learning by Sebastian Raschka（Packt Publishing）

·Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron（O’Reilly）

虽然书籍可以成为宝贵的学习资源，但当底层开源软件发生变化时，它们有时会变得过时。熟悉各种统计或机器学习框架的文档以了解最新功能和API是个不错的主意。

第14章　数据分析示例

现在我们已经到达本书的主要章节的尾部，我们将试一些真实世界的数据集。对于每个数据集，我们将使用本书中提到的技术，从数据中提取数据的意义。已经介绍的技术可以被应用到所有其他数据集上，包括你自己的数据集。本章包含各种各样的示例数据集，你可以使用本书中的工具练习这些示例数据集。

示例数据集可在本书附带的GitHub仓库（http://github.com/wesm/pydata-book ）中找到。

14.1 从Bitly获取1.USA.gov数据

2011年，短网址服务商Bitly（https://bitly.com/ ）与美国政府网站USA.gov（https://www.usa.gov/ ）合作，提供从以.gov或.mil结尾的短网址的用户收集的匿名数据。2011年，可以下载的文本文件提供实时供稿和小时快照。在撰写本书时（2017年）此服务已关闭，但我们保留了一个数据文件作为本书的示例。

在每小时快照的情况下，每个文件中的每一行都包含一种通用形式的web数据，称为JSON，它是JavaScript Object Notation的简写。例如，如果我们只读取文件的第一行，我们可能会看到类似这样的内容：

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US"
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf"
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf",
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 13319232
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

Python拥有用于将JSON字符串转换为Python字典对象的内置和第三方库。这里我们将在我们下载的示例文件的每一行中使用json模块及其loads函数：

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

结果对象records现在是一个Python字典的列表：

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML,
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
```

```
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

14.1.1 纯Python时区计数

假设我们想要找到数据集中最常出现的时区（tz字段）。有很多方法可以实现这一点。首先，我们使用列表再次提取时区列表：

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

哇！结果并不是所有的记录都有时区字段。这很容易处理，因为我们可以在列表推导式的结尾添加一个检查if'tz'in rec：

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

只看前10个时区，我们可以看到其中一些是未知的（空字符串）。你也可以过滤掉这些，但我现在就把它们留下。现在，为了按时区生成计数，我将展示两种方法：更难的方法（仅使用Python标准库）和更简单的方法（使用pandas）。计数的一种方法是在遍历时区时使用字典来存储计数：

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
```

```
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

使用更多Python标准库中的高级工具，你可以用更简明的方式写出同样的东西：

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # 值将会初始化为0
    for x in sequence:
        counts[x] += 1
    return counts
```

我把这个逻辑放在一个函数中，使它有更好的可重用性。要在时区上使用它，只需传递time_zones列表：

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

如果我们想要前十的时区和它们的计数，我们可以做一些字典技巧：

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

之后我们有：

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
```

```
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

如果我们搜索Python标准库，你可能会发现collections.Counter类，它可以使任务更加简单：

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

14.1.2　使用pandas进行时区计数

根据原始的记录集合生成DataFrame非常简单，只需要把记录的列表传递给pandas.DataFrame：

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_    120 non-null float64
a              3440 non-null object
al             3094 non-null object
c              2919 non-null object
cy             2919 non-null object
g              3440 non-null object
gr             2919 non-null object
h              3440 non-null object
hc             3440 non-null float64
hh             3440 non-null object
kw             93 non-null object
l              3440 non-null object
ll             2919 non-null object
nk             3440 non-null float64
r              3440 non-null object
t              3440 non-null float64
tz             3440 non-null object
u              3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
6        Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

frame的输出显示的是概要视图，用于展示大型DataFrame对象。对于Series，我们可以使用value_counts方法：

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
America/Sao_Paulo        33
Name: tz, dtype: int64
```

我们可以使用matplotlib对这些数据可视化。你可以进行一些清理工作，以便为记录中的未知和缺失的时区数据填入替代值。我们用fillna方法替换缺失值，并为空字符串使用布尔数组索引：

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'
In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
America/New_York       1251
Unknown                 521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Missing                 120
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
Name: tz, dtype: int64
```

在这里，我们可以使用seaborn包（http://seaborn.pydata.org/ ）来绘制一个水平柱状图（见图14-1的可视化结果）：

```
In [36]: import seaborn as sns
```

```
In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

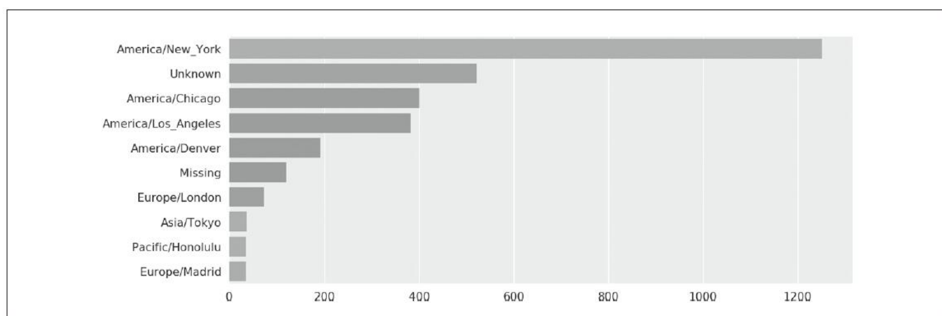


图14-1：1.usa.gov样本数据的时区TOP计数

a列包含了执行网址缩短的浏览器、设备或应用的信息：

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Fire

In [41]: frame['a'][51][:50]  # 很长的一行
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

解析这些“代理”字符串中的所有感兴趣的信息可能看起来像是一项艰巨的任务。一种可能的策略是分离字符串中的第一个标记（粗略地对应于浏览器功能），并对用户行为进行另一个概括：

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.dropna()

In [43]: results[:5]
Out[43]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
```

```
Out[44]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
dtype: int64
```

现在，假设您想将时区计数多的时区记录分解为Windows和非Windows用户。作为简化，如果字符串'Windows'在代理理字符串中，我们就认为用户在Windows上。由于一些代理理字符串的缺失，我们将从数据中排除这些代理理字符串：

```
In [45]: cframe = frame[frame.a.notnull()]
```

之后我们想要计算一个代表每一行是否是Windows的值：

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
   ....:                         'Windows', 'Not Windows')

In [48]: cframe['os'][:5]
Out[48]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: os, dtype: object
```

之后，你可以根据时区列以及新生成的操作系统列对数据进行分组：

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

与value_counts函数类似，分组计数可以使用size计算。然后可以使用unstack对计算结果进行重塑：

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
```

```
Out[51]:
os                                 Not Windows  Windows
tz
                                         245.0    276.0
Africa/Cairo                               0.0      3.0
Africa/Casablanca                          0.0      1.0
Africa/Ceuta                               0.0      2.0
Africa/Johannesburg                        0.0      1.0
Africa/Lusaka                              0.0      1.0
America/Anchorage                          4.0      1.0
America/Argentina/Buenos_Aires             1.0      0.0
America/Argentina/Cordoba                  0.0      1.0
America/Argentina/Mendoza                  0.0      1.0
```

最后，让我们选出总体计数最高的时区。要实现这个功能，我在agg_counts中根据行的计数构造了一个间接索引数组：

```
# 用于升序排列

In [52]: indexer = agg_counts.sum(1).argsort()

In [53]: indexer[:10]
Out[53]:
tz
                                  24
Africa/Cairo                      20
Africa/Casablanca                 21
Africa/Ceuta                      92
Africa/Johannesburg               87
Africa/Lusaka                     53
America/Anchorage                 54
America/Argentina/Buenos_Aires    57
America/Argentina/Cordoba         26
America/Argentina/Mendoza         55
dtype: int64
```

我使用take方法按顺序选出行，之后再对最后10行进行切片（最大的10个值）：

```
In [54]: count_subset = agg_counts.take(indexer[-10:])

In [55]: count_subset
Out[55]:
os                    Not Windows  Windows
tz
America/Sao_Paulo            13.0     20.0
Europe/Madrid                16.0     19.0
Pacific/Honolulu              0.0     36.0
```

```
Asia/Tokyo                        2.0       35.0
Europe/London                    43.0       31.0
America/Denver                  132.0       59.0
America/Los_Angeles             130.0      252.0
America/Chicago                 115.0      285.0
                                245.0      276.0
America/New_York                339.0      912.0
```

pandas有一个便捷的方法叫作nlargest，可以做同样的事情：

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York       1251.0
                        521.0
America/Chicago         400.0
America/Los_Angeles     382.0
America/Denver          191.0
Europe/London            74.0
Asia/Tokyo               37.0
Pacific/Honolulu         36.0
Europe/Madrid            35.0
America/Sao_Paulo        33.0
dtype: float64
```

然后，如前面的代码块所示，这些数据可以绘制在条形图中；通过向seaborn的barplot函数传递一个额外的参数，我将绘制一个堆积条形图（参见图14-2）：

```
# 对绘图数据重新排列
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                  tz           os  total
0  America/Sao_Paulo  Not Windows   13.0
1  America/Sao_Paulo      Windows   20.0
2      Europe/Madrid  Not Windows   16.0
3      Europe/Madrid      Windows   19.0
4   Pacific/Honolulu  Not Windows    0.0
5   Pacific/Honolulu      Windows   36.0
6         Asia/Tokyo  Not Windows    2.0
7         Asia/Tokyo      Windows   35.0
8      Europe/London  Not Windows   43.0
```

```
9       Europe/London       Windows    31.0

In [62]: sns.barplot(x='total', y='tz', hue='os',  data=count_subset
```

该图不容易看到较小组中的Windows用户的相对百分比，因此让我们将组百分比归一化为1：

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

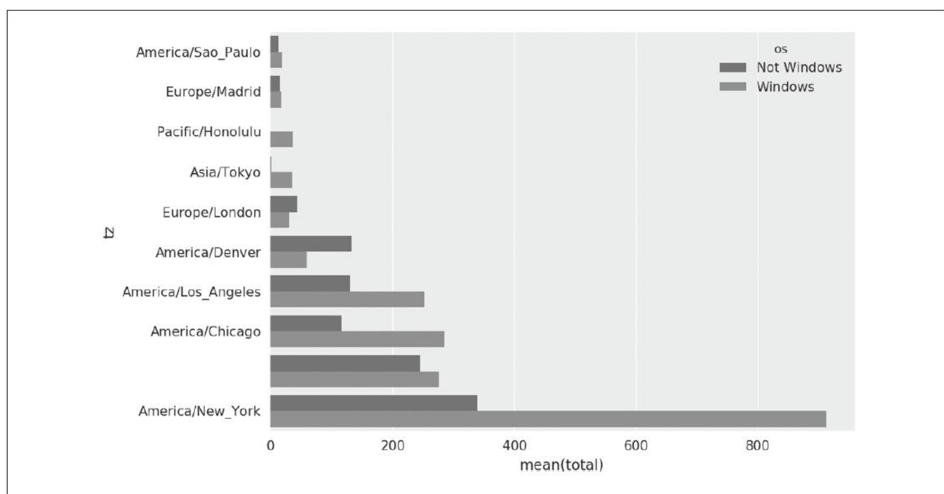


图14-2：按是否是Windows用户划分的最高计数时区

然后在图14-3中绘图：

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os',  data=resul
```

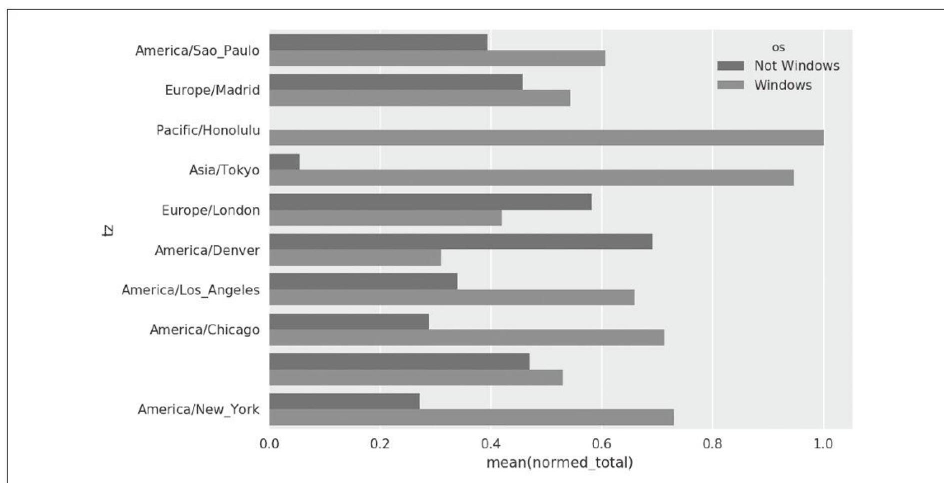


图14-3：按是否是Windows用户划分的最高计数时区百分比

我们可以通过transform方法和groupby方法更有效地计算归一化之和：

```
In [66]: g = count_subset.groupby('tz')
In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 MovieLens 1M数据集

GroupLens实验室（http://www.grouplens.org/node/73 ）提供了一些从MovieLens用户那里收集的20世纪90年代末和21世纪初的电影评分数据的集合。这些数据提供了电影的评分、电影的元数据（流派和年份）以及观众数据（年龄、邮编、性别、职业）。这些数据通常会用于基于机器学习算法的推荐系统开发，虽然我们不会在本书中详细探讨机器学习技术，但我会向你展示如何将这些数据集切片并切成你需要的确切形式。

MovieLens 1M数据集包含6，000个用户对4，000部电影的100万个评分。数据分布在三个表格中：评分，用户信息和电影信息。从ZIP文件中提取数据后，我们可以使用pandas.read_table将每个表加载到一个pandas DataFrame对象中：

```
import pandas as pd

# 让展示内容少一点
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::',
                      header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',
                        header=None, names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       header=None, names=mnames)
```

你可以通过使用Python的切片语法来查看每个DataFrame的前几行来验证一切是否成功：

```
In [69]: users[:5]
Out[69]:
   user_id gender  age  occupation    zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [70]: ratings[:5]
```

```
Out[70]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291

In [71]: movies[:5]
Out[71]:
   movie_id                               title
0         1                    Toy Story (1995)   Animation|Children
1         2                      Jumanji (1995)  Adventure|Children's
2         3             Grumpier Old Men (1995)                Comedy
3         4            Waiting to Exhale (1995)                  Come
4         5  Father of the Bride Part II (1995)

In [72]: ratings
Out[72]:
         user_id  movie_id  rating  timestamp
0              1      1193       5  978300760
1              1       661       3  978302109
2              1       914       3  978301968
3              1      3408       4  978300275
4              1      2355       5  978824291
...          ...       ...     ...        ...
1000204     6040      1091       1  956716541
1000205     6040      1094       5  956704887
1000206     6040       562       5  956704746
1000207     6040      1096       4  956715648
1000208     6040      1097       4  956715569
[1000209 rows x 4 columns]
```

请注意，年龄和职业被编码为整数，这些表示了数据集的README文件所描述的分组。跨越三个表格分析数据并不是一件简单的事情，例如，假设你想按性别和年龄计算某个电影的平均评分。正如你将看到的，将所有表格合并到单个表中会更容易。使用pandas的合并功能，我们首先将ratings表与users表合并，然后将该结果与movies表数据合并。pandas根据重叠名称推断哪些列用作合并的（或连接）键位：

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)
In [74]: data
Out[74]:
         user_id  movie_id  rating  timestamp gender  age  occupation
0              1      1193       5  978300760      F    1          10
1              2      1193       5  978298413      M   56          16
2             12      1193       4  978220179      M   25          12
3             15      1193       4  978199279      M   25           7
4             17      1193       5  978158471      M   50           1
...          ...       ...     ...        ...    ...  ...         ...
```

```
1000204    5949    2198    5  958846401    M  18      1
1000205    5675    2703    3  976029116    M  35      1
1000206    5780    2845    1  958153068    M  18      1
1000207    5851    3607    5  957756608    F  18      2
1000208    5938    2909    4  957273353    M  25
                                              title
0             One Flew Over the Cuckoo's Nest (1975)
1             One Flew Over the Cuckoo's Nest (1975)
2             One Flew Over the Cuckoo's Nest (1975)
3             One Flew Over the Cuckoo's Nest (1975)
4             One Flew Over the Cuckoo's Nest (1975)
...                                             ...
1000204                       Modulations (1998)           Docume
1000205                    Broken Vessels (1998)
1000206                        White Boys (1999)
1000207                 One Little Indian (1973)  Comedy|Drama|We
1000208  Five Wives, Three Secretaries and Me (1998)        Docume
[1000209 rows x 10 columns]
In [75]: data.iloc[0]
Out[75]:
user_id                                         1
movie_id                                     1193
rating                                          5
timestamp                               978300760
gender                                          F
age                                             1
occupation                                     10
zip                                         48067
title        One Flew Over the Cuckoo's Nest (1975)
genres                                      Drama
Name: 0, dtype: object
```

为了获得按性别分级的每部电影的平均电影评分，我们可以使用pivot_table方法：

```
In [76]: mean_ratings = data.pivot_table('rating', index='title',
   ....:                                 columns='gender', aggfunc='r

In [77]: mean_ratings[:5]
Out[77]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)  3.828571  3.689024
```

上面的代码产生了另一个DataFrame，其中包含电影标题作为行标签

（“索引”）和性别作为列标签的平均评分。我首先过滤掉少于250（完全随意定的数字）个评分的电影；为此，我接着按标题对数据进行分组，并使用size（）为每个标题获取一个元素是各分组大小的Series：

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
title
$1,000,000 Duck (1971)                37
'Night Mother (1986)                  70
'Til There Was You (1997)             52
'burbs, The (1989)                   303
...And Justice for All (1979)        199
1-900 (1994)                           2
10 Things I Hate About You (1999)    700
101 Dalmatians (1961)                565
101 Dalmatians (1996)                364
12 Angry Men (1957)                  616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratings_by_title >= 2

In [81]: active_titles
Out[81]:
Index([''burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Me
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey
       '2010 (1984)',
       ...
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

评分多于250个的电影标题的索引之后可以用于从mean_ratings中选出所需的行：

```
# 在牵引上选取行
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
gender                                    F         M
title
'burbs, The (1989)                 2.793478  2.962085
```

```
10 Things I Hate About You (1999)  3.646552  3.311966
101 Dalmatians (1961)              3.791444  3.500000
101 Dalmatians (1996)              3.240000  2.911215
12 Angry Men (1957)                4.184397  4.328421
...                                     ...       ...
Young Guns (1988)                  3.371795  3.425620
Young Guns II (1990)               2.934783  2.904025
Young Sherlock Holmes (1985)       3.514706  3.363344
Zero Effect (1998)                 3.864407  3.723140
eXistenZ (1999)                    3.098592  3.289086
[1216 rows x 2 columns]
```

要看到女性观众的top电影，我们可以按F列降序排序：

```
In [85]: top_female_ratings = mean_ratings.sort_values(by='F', ascend

In [86]: top_female_ratings[:10]
Out[86]:
gender                                                         F
title
Close Shave, A (1995)                                   4.644444  4.4737
Wrong Trousers, The (1993)                              4.588235  4.4782
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)           4.572650  4.4645
Wallace & Gromit: The Best of Aardman Animation...      4.563107  4.3850
Schindler's List (1993)                                 4.562602  4.4914
Shawshank Redemption, The (1994)                        4.539075  4.5606
Grand Day Out, A (1992)                                 4.537879  4.2932
To Kill a Mockingbird (1962)                            4.536667  4.3726
Creature Comforts (1990)                                4.513889  4.2722
Usual Suspects, The (1995)                              4.513317  4.5182
```

14.2.1 测量评价分歧

假设你想找到男性和女性观众之间最具分歧性的电影。一种方法是添加一列到含有均值差的mean_ratings中，然后按以下方式排序：

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F'
```

按照'diff'排序产生评分差异最大的电影，以便我们可以看到哪些是女性首选的：

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by='diff')

In [89]: sorted_by_diff[:10]
Out[89]:
gender                                        F         M      diff
title
Dirty Dancing (1987)                   3.790378  2.959596 -0.830782
Jumpin' Jack Flash (1986)              3.254717  2.578358 -0.676359
Grease (1978)                          3.975265  3.367041 -0.608224
Little Women (1994)                    3.870588  3.321739 -0.548849
Steel Magnolias (1989)                 3.901734  3.365957 -0.535777
Anastasia (1997)                       3.800000  3.281609 -0.518391
Rocky Horror Picture Show, The (1975)  3.673016  3.160131 -0.512885
Color Purple, The (1985)               4.158192  3.659341 -0.498851
Age of Innocence, The (1993)           3.827068  3.339506 -0.487561
Free Willy (1993)                      2.921348  2.438776 -0.482573
```

转换行的顺序，并切片出top 10的行，我们就可以获得男性更喜欢但女性评分不高的电影：

```
# 对行倒序，取前10行
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
gender                                         F         M      diff
title
Good, The Bad and The Ugly, The (1966)  3.494949  4.221300  0.726351
Kentucky Fried Movie, The (1977)        2.878788  3.555147  0.676359
Dumb & Dumber (1994)                    2.697987  3.336595  0.638608
Longest Day, The (1962)                 3.411765  4.031447  0.619682
Cable Guy, The (1996)                   2.250000  2.863787  0.613787
Evil Dead II (Dead By Dawn) (1987)      3.297297  3.909283  0.611985
Hidden, The (1987)                      3.137931  3.745098  0.607167
Rocky III (1982)                        2.361702  2.943503  0.581801
```

```
Caddyshack (1980)                          3.396135  3.969737  0.573602
For a Few Dollars More (1965)              3.409091  3.953795  0.544704
```

假设你想要的是不依赖于性别标识而在观众中引起最大异议的电影。异议可以通过评分的方差或标准差来衡量。

```
# Standard deviation of rating grouped by title
In [91]: rating_std_by_title = data.groupby('title')['rating'].std()

# Filter down to active_titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_titles]

# Order Series by value in descending order
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
title
Dumb & Dumber (1994)                     1.321333
Blair Witch Project, The (1999)          1.316368
Natural Born Killers (1994)              1.307198
Tank Girl (1995)                         1.277695
Rocky Horror Picture Show, The (1975)    1.260177
Eyes Wide Shut (1999)                    1.259624
Evita (1996)                             1.253631
Billy Madison (1995)                     1.249970
Fear and Loathing in Las Vegas (1998)    1.246408
Bicentennial Man (1999)                  1.245533
Name: rating, dtype: float64
```

你可能已经注意到，电影流派是以管道分隔（|）字符串的形式给出的。如果你想按流派做一些分析，需要做更多的工作来将流派信息转化为更有用的形式。

14.3　美国1880～2010年的婴儿名字

美国社会保障局（SSA）提供了从1880年至现在的婴儿姓名频率的数据。Hadley Wickham是一些流行R包的作者，他经常利用这个数据集来说明R中的数据操作。

我们需要做一些数据规整来加载这个数据集，但一旦我们这样做，我们将得到一个看起来如下的DataFrame：

```
In [4]: names.head(10)
Out[4]:
        name sex  births  year
0       Mary   F    7065  1880
1       Anna   F    2604  1880
2       Emma   F    2003  1880
3  Elizabeth   F    1939  1880
4     Minnie   F    1746  1880
5   Margaret   F    1578  1880
6        Ida   F    1472  1880
7      Alice   F    1414  1880
8     Bertha   F    1320  1880
9      Sarah   F    1288  1880
```

你可能想要对数据集做很多事情：

·根据给定的名字（可以是你自己的，或另一个名字）对婴儿名字随时间的比例进行可视化

·确定一个名字的相对排位

·确定每年最受欢迎的名字，或者流行程度最高或最低的名字

·分析名字趋势：元音、辅音、长度、整体多样性、拼写变化、第一个和最后一个字母

·分析额外的趋势来源：圣经中的名字，名人，人口变化

通过本书中的工具，许多这些分析都可以实现，所以我会引导你了解其中的一部分。

截至撰写本书时，美国社会保障局每年提供一份数据文件，其中包含每个性别/姓名组合的出生总数。这些文件的原始档案可以从http://

www.ssa.gov/oact/babynames/limits.html 获得。

如果你在阅读本书时，上面的网页被移动了，你可以再次通过互联网搜索找到该页面。下载“国家数据”文件names.zip并解压缩，你将拥有一个包含一系列文件（如yob1880.txt）的目录。我使用Unix head命令查看其中一个文件的前10行（在Windows上，可以使用more命令或在文本编辑器中打开它）：

```
In [94]: !head -n 10 datasets/babynames/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

由于它已经以逗号分隔的形式出现，因此可以使用pandas.read_csv将其加载到DataFrame中：

```
In [95]: import pandas as pd

In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
   ....:                         names=['name', 'sex', 'births'])

In [97]: names1880
Out[97]:
           name sex  births
0          Mary   F    7065
1          Anna   F    2604
2          Emma   F    2003
3     Elizabeth   F    1939
4        Minnie   F    1746
...         ...  ..     ...
1995     Woodie   M       5
1996     Worthy   M       5
1997     Wright   M       5
1998       York   M       5
1999  Zachariah   M       5
[2000 rows x 3 columns]
```

这些文件只包含每年至少有五次出现的名字，为简单起见，我们可以使用按性别列出的出生总和作为当年的出生总数：

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F     90993
M    110493
Name: births, dtype: int64
```

由于数据集按年分为多个文件，首先要做的事情之一是将所有数据集中到一个DataFrame中，然后再添加一个年份字段。你可以使用pandas.concat来做到这一点：

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'datasets/babynames/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# 将所有内容粘接进一个DataFrame
names = pd.concat(pieces, ignore_index=True)
```

这里有几件事要注意。首先，请记住，concat默认情况下将DataFrame对象以逐行方式黏合在一起。其次，你必须传递ignore_index=True，因为我们不想保留从read_csv返回的原始行号。所以我们现在有一个非常大的包含所有名字数据的DataFrame：

```
In [100]: names
Out[100]:
              name sex  births  year
0             Mary   F    7065  1880
1             Anna   F    2604  1880
2             Emma   F    2003  1880
3        Elizabeth   F    1939  1880
4           Minnie   F    1746  1880
...            ...  ..     ...   ...
1690779    Zymaire   M       5  2010
1690780     Zyonne   M       5  2010
1690781  Zyquarius   M       5  2010
1690782      Zyran   M       5  2010
1690783      Zzyzx   M       5  2010
[1690784 rows x 4 columns]
```

掌握这些数据后，我们可以使用groupby或pivot_table开始聚合年份和性别的数据（参见图14-4）：

```
In [101]: total_births = names.pivot_table('births', index='year',
   .....:                                  columns='sex', aggfunc=sur

In [102]: total_births.tail()
Out[102]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [103]: total_births.plot(title='Total births by sex and year')
```

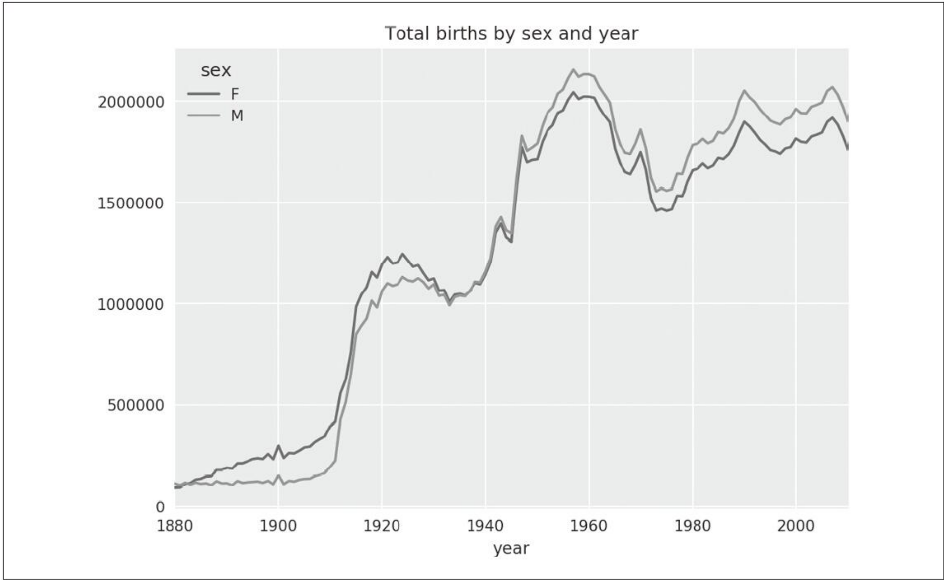


图14-4：按性别和年份划分的出生总数

接下来，让我们插入一个prop列，给出每个婴儿名字相对于出生总数的比例。prop值为0.02表示每100个婴儿中有2个起了某个名字。因此，我们按年份和性别对数据进行分组，然后将新列添加到每个组：

```
def add_prop(group):
    group['prop'] = group.births / group.births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

生成的完整数据集现在具有以下列：

```
In [105]: names
Out[105]:
                name sex  births  year      prop
0               Mary   F    7065  1880  0.077643
1               Anna   F    2604  1880  0.028618
2               Emma   F    2003  1880  0.022013
3          Elizabeth   F    1939  1880  0.021309
4            Minnie   F    1746  1880  0.019188
...              ...  ..     ...   ...       ...
1690779      Zymaire   M       5  2010  0.000003
1690780       Zyonne   M       5  2010  0.000003
1690781    Zyquarius   M       5  2010  0.000003
1690782        Zyran   M       5  2010  0.000003
1690783        Zzyzx   M       5  2010  0.000003
[1690784 rows x 5 columns]
```

在执行此类组操作时，进行完整性检查通常很有价值，例如验证所有组中的prop列总计为1：

```
In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year  sex
1880  F      1.0
      M      1.0
1881  F      1.0
      M      1.0
1882  F      1.0
            ...
2008  M      1.0
2009  F      1.0
      M      1.0
2010  F      1.0
      M      1.0
Name: prop, Length: 262, dtype: float64
```

现在完成了，我将提取一部分数据以便于进一步分析：每个性别/年份组合的前1，000名。这是另一个小组操作：

```
def get_top1000(group):
    return group.sort_values(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# 删除组索引，不需要它
top1000.reset_index(inplace=True, drop=True)
```

如果你喜欢DIY方式，可以试试以下代码：

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_values(by='births', ascending=False)[:1
top1000 = pd.concat(pieces, ignore_index=True)
```

结果数据集现在比较小：

```
In [108]: top1000
Out[108]:
               name sex  births  year      prop
0              Mary   F    7065  1880  0.077643
1              Anna   F    2604  1880  0.028618
2              Emma   F    2003  1880  0.022013
3         Elizabeth   F    1939  1880  0.021309
4            Minnie   F    1746  1880  0.019188
...             ...  ..     ...   ...       ...
261872       Camilo   M     194  2010  0.000102
261873       Destin   M     194  2010  0.000102
261874       Jaquan   M     194  2010  0.000102
261875       Jaydan   M     194  2010  0.000102
261876       Maxton   M     193  2010  0.000102
[261877 rows x 5 columns]
```

我们将在后续的数据调查中使用这个Top 1，000数据集。

14.3.1 分析名字趋势

利用完整的数据集和手上的Top 1，000数据集，我们可以开始分析各种感兴趣的命名趋势。首先，将Top 1，000的名字分成男孩和女孩两部分是很容易实现的：

```
In [109]: boys = top1000[top1000.sex == 'M']

In [110]: girls = top1000[top1000.sex == 'F']
```

简单的时间序列，比如每年的John或Marry的数量，都可以绘制出来，但是需要一些处理才能更有用。让我们按年份和名字形成出生总数的数据透视表：

```
In [111]: total_births = top1000.pivot_table('births', index='year',
   .....:                                     columns='name',
   .....:                                     aggfunc=sum)
```

现在，可以使用DataFrame的plot方法绘制少数名称的透视表（图14-5显示了结果）：

```
In [112]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB

In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   .....:             title="Number of births per year")
```

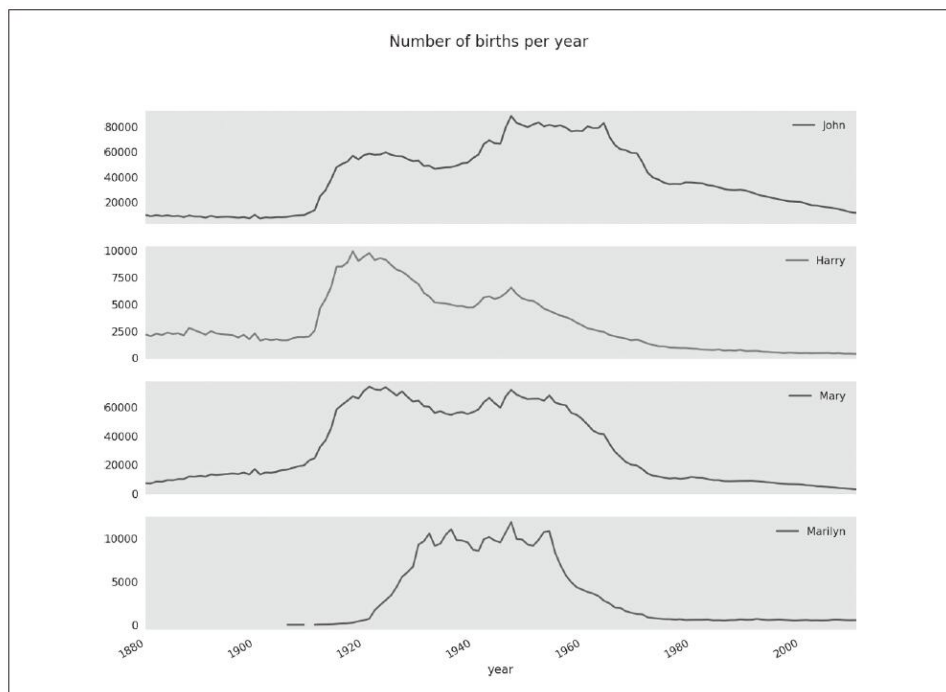


图14-5：一些男孩名字和女孩名字随时间变化的趋势

看着上图，你可能会得出结论，这些名字已经不适合美国人了。但是故事实际上比这更复杂，下一节将会探讨这个故事。

14.3.1.1　计量命名多样性的增加

对上一节中趋势下降的一个解释是，越来越多的父母为他们的孩子选择常用名字。这个假设可以在数据中进行探索和确认。其中一个衡量指标是Top 1，000最受欢迎的名字所涵盖婴儿的出生比例，我按照年份和性别进行聚合和绘图（图14-6显示了结果图）：

```
In [116]: table = top1000.pivot_table('prop', index='year',
   .....:                              columns='sex', aggfunc=sum)

In [117]: table.plot(title='Sum of table1000.prop by year and sex',
   .....:            yticks=np.linspace(0, 1.2, 13), xticks=range(188
)
```

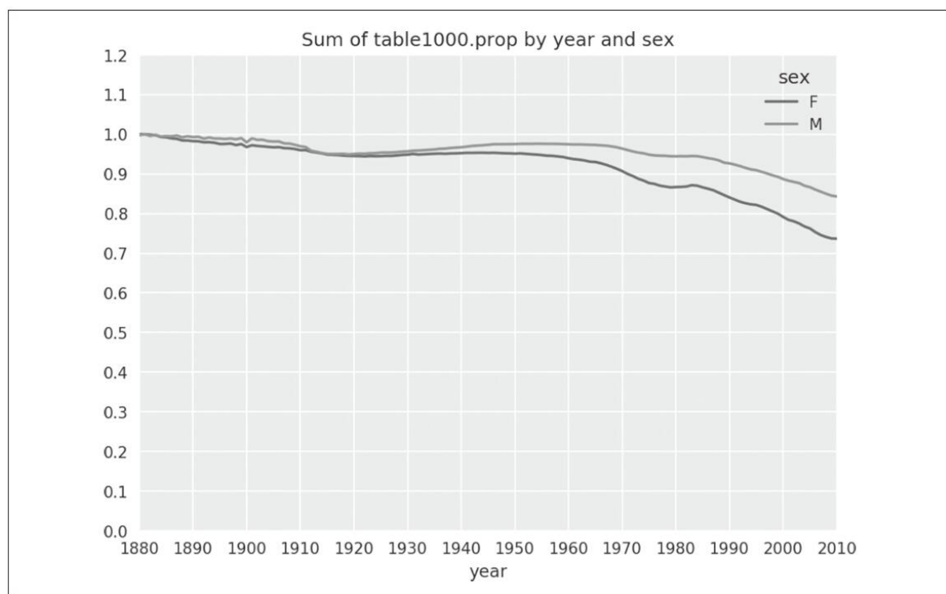


图14-6：按性别划分的Top 1000名字的出生比例

你可以看到，事实上似乎有越来越多的名字多样性（Top 1，000名字的总比例的降低）。另一个有趣的指标是不同名字的数量，按最高到最低的受欢迎程度在出生人数最高的50%的名字中排序。这个数字计算起来有点棘手。让我们考虑一下2010年的男孩名字：

```
In [118]: df = boys[boys.year == 2010]

In [119]: df
Out[119]:
            name  sex  births  year      prop
260877     Jacob    M   21875  2010  0.011523
260878     Ethan    M   17866  2010  0.009411
260879   Michael    M   17133  2010  0.009025
260880    Jayden    M   17030  2010  0.008971
260881   William    M   16870  2010  0.008887
...          ...   ..     ...   ...       ...
261872    Camilo    M     194  2010  0.000102
261873    Destin    M     194  2010  0.000102
261874    Jaquan    M     194  2010  0.000102
261875    Jaydan    M     194  2010  0.000102
261876    Maxton    M     193  2010  0.000102
[1000 rows x 5 columns]
```

按降序排列prop后，我们想知道有多少名字是最受欢迎的那50%。你

可以写一个for循环来实现这一点，但矢量化的NumPy方式更聪明一些。获取prop的累积总和cumsum，然后调用searchsorted方法返回累积总和中的位置，在该处需要插入0.5以保持排序顺序：

```
In [120]: prop_cumsum = df.sort_values(by='prop', ascending=False).p

In [121]: prop_cumsum[:10]
Out[121]:
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

由于数组是零索引的，所以给这个结果加1会得到117的结果。相比之下，在1900年这个数字要小得多：

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).prop.c

In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

你现在可以将此操作应用于每个年/性别分组，通过这些字段进行groupby，并将返回值是每个分组计数值的函数apply到每个分组上：

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_coun
diversity = diversity.unstack('sex')
```

产生的DataFrame diversity现在有两个时间序列，每个时间序列对应

一种性别，并按照年份索引。diversity像之前一样可以在IPython中被检查并绘制（见图14-7）：

```
In [128]: diversity.head()
Out[128]:
sex     F   M
year
1880   38  14
1881   38  14
1882   38  15
1883   39  15
1884   39  16

In [129]: diversity.plot(title="Number of popular names in top 50%")
```

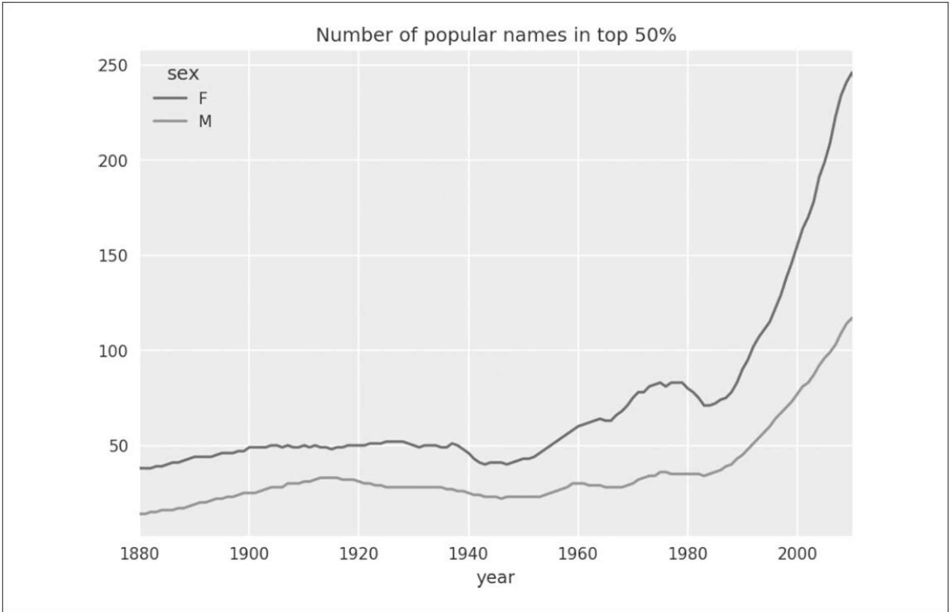


图14-7：按年份划分的多样性指标图

正如你所看到的，女孩的名字一直比男孩的名字更加多样化，而且随着时间的推移它们变得越来越多。读者可以进一步分析究竟是什么促成了多样性，比如增加了其他拼写方法。

14.3.1.2 “最后一个字母”革命

2007年，婴儿名字研究员Laura Wattenberg在她的网站（http://

www.babynamewizard.com ）上指出，男孩名字最后一个字母的分布在过去的一百年里发生了重大变化。为了看到这一点，我们首先按照年份、性别和最后一个字母汇总完整数据集中的所有出生情况：

```
# 从name列提取最后一个字母
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', index=last_letters,
                          columns=['sex', 'year'], aggfunc=sum)
```

然后，我们选出历史上三个有代表性的年份并列出前几行：

```
In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level=

In [132]: subtable.head()
Out[132]:
sex                 F                            M
year             1910      1960      2010     1910      1960      20
last_letter
a            108376.0  691247.0  670605.0    977.0    5204.0    28438
b                 NaN     694.0     450.0    411.0    3912.0    38859
c                 5.0      49.0     946.0    482.0   15476.0    23125
d              6750.0    3729.0    2607.0  22111.0  262112.0    44398
e            133569.0  435013.0  313833.0  28655.0  178823.0   129012
```

接下来，按照出生总数对表格进行归一化处理，计算一个新表格，其中包含每个性别的每个结束字母占总出生数的比例：

```
In [133]: subtable.sum()
Out[133]:
sex  year
F    1910     396416.0
     1960    2022062.0
     2010    1759010.0
M    1910     194198.0
     1960    2132588.0
     2010    1898382.0
dtype: float64

In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
sex                 F                              M
```

```
year             1910      1960      2010      1910      1960      20
last_letter
a            0.273390  0.341853  0.381240  0.005031  0.002440  0.014
b                 NaN  0.000343  0.000256  0.002116  0.001834  0.020
c            0.000013  0.000024  0.000538  0.002482  0.007257  0.012
d            0.017028  0.001844  0.001482  0.113858  0.122908  0.023
e            0.336941  0.215133  0.178415  0.147556  0.083853  0.067
...               ...       ...       ...       ...       ...
v                 NaN  0.000060  0.000117  0.000113  0.000037  0.001
w            0.000020  0.000031  0.001182  0.006329  0.007711  0.016
x            0.000015  0.000037  0.000727  0.003965  0.001851  0.008
y            0.110972  0.152569  0.116828  0.077349  0.160987  0.058
z            0.002439  0.000659  0.000704  0.000170  0.000184  0.001
[26 rows x 6 columns]
```

现在根据掌握的字母比例，我们可以绘制出按年划分的每个性别的条形图（见图14-8）：

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)
```

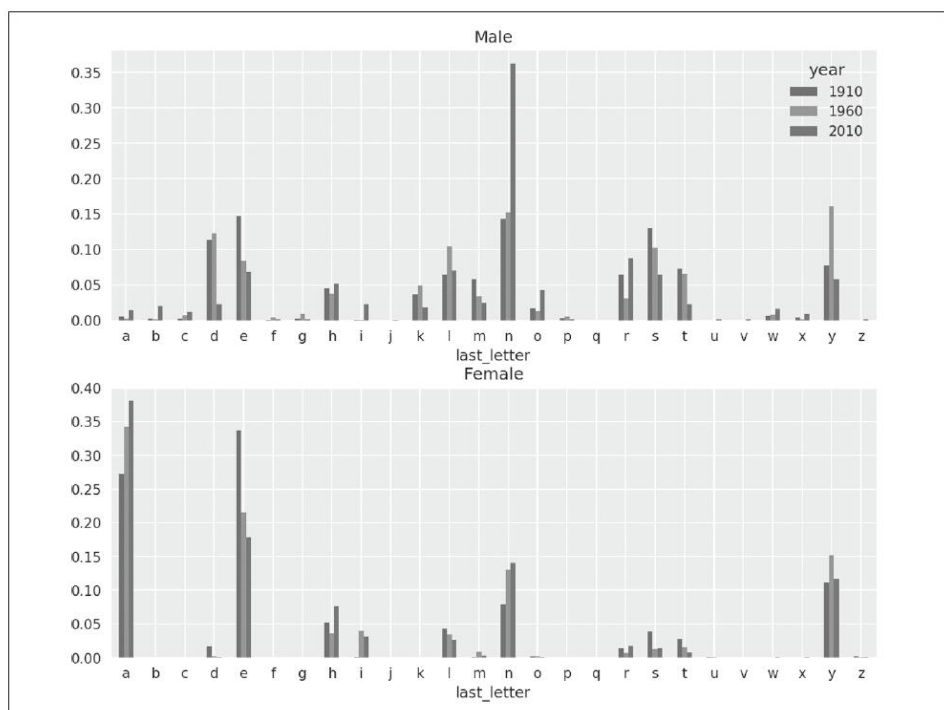


图14-8：男孩和女孩名字最后一个字母的比例

如你所见，自20世纪60年代以来，以n结尾的男孩名字经历了显著的增长。回到之前创建的完整表格，我再次按年份和性别进行标准化，并为男孩名字选择一个字母子集，最后转换为使每列成为一个时间序列：

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.loc[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:
last_letter          d         n         y
year
1880          0.083055  0.153213  0.075760
1881          0.083247  0.153214  0.077451
1882          0.085340  0.149560  0.077537
1883          0.084066  0.151646  0.079144
1884          0.086120  0.149915  0.080405
```

根据掌握的时间序列DataFrame，我可以使用plot方法绘制字母随时

间变化的趋势（见图14-9）：

```
In [143]: dny_ts.plot()
```

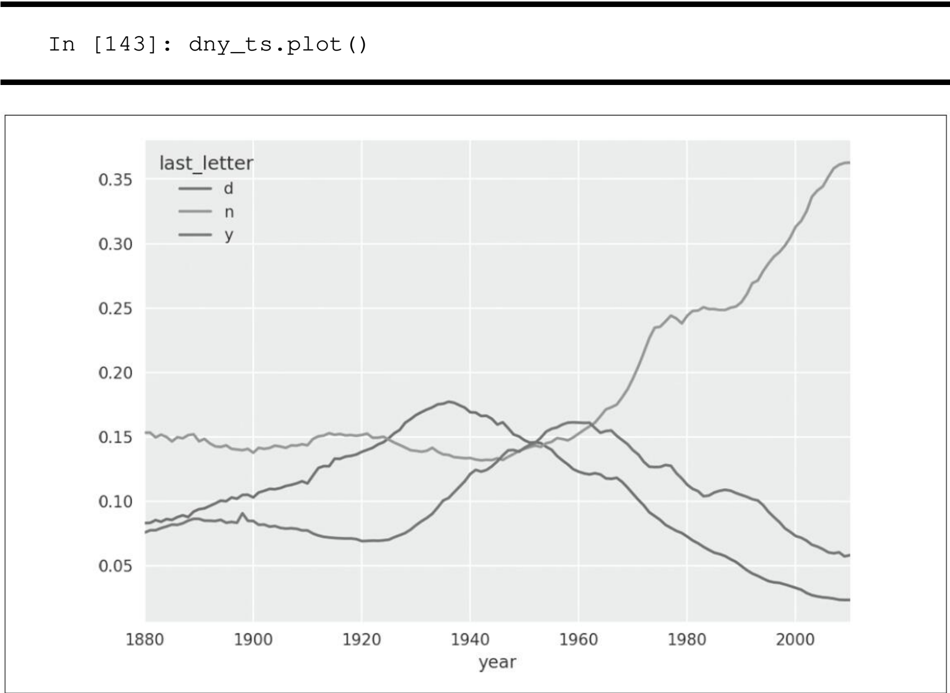


图14-9：随着时间推移名字以d/n/y结尾的男孩的比例变化趋势

14.3.1.3　男孩名字变成女孩名字（以及反向）

另一个有趣的趋势是看到样本中较早在男性中流行的男孩名字，但现在已经“改变了性别”。一个例子是Lesley或Leslie的名字。回到top1000的DataFrame，我计算数据集中以“lesl”开头的名字列表：

```
In [144]: all_names = pd.Series(top1000.name.unique())

In [145]: lesley_like = all_names[all_names.str.lower().str.contains

In [146]: lesley_like
Out[146]:
632     Leslie
2294    Lesley
4262    Leslee
4728     Lesli
6103     Lesly
dtype:  object
```

从DataFrame那里，我们可以过滤掉那些名字，并对按名字分组的出生数进行累加来看相关频率：

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee      1082
Lesley     35022
Lesli        929
Leslie    370429
Lesly      10067
Name: births, dtype: int64
```

之后，让我们按性别和年份进行聚合，并在年内进行标准化：

```
In [149]: table = filtered.pivot_table('births', index='year',
   .....:                              columns='sex', aggfunc='sum')

In [150]: table = table.div(table.sum(1), axis=0)

In [151]: table.tail()
Out[151]:
sex     F   M
year
2006  1.0 NaN
2007  1.0 NaN
2008  1.0 NaN
2009  1.0 NaN
2010  1.0 NaN
```

最后我们我们可以绘制出按性别随时间推移的分解图（图14-10）：

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

14.4 美国农业部食品数据库

美国农业部（US Department of Agriculture，USDA）提供了食物营养信息数据库。程序员Ashley Williams以JSON格式提供了这个数据库的一个版本。记录如下所示：

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
    ...

  ],
  "nutrients": [
    {
      "value": 20.8,
      "units": "g",
      "description": "Protein",
      "group": "Composition"
    },

    ...
  ]
}
```

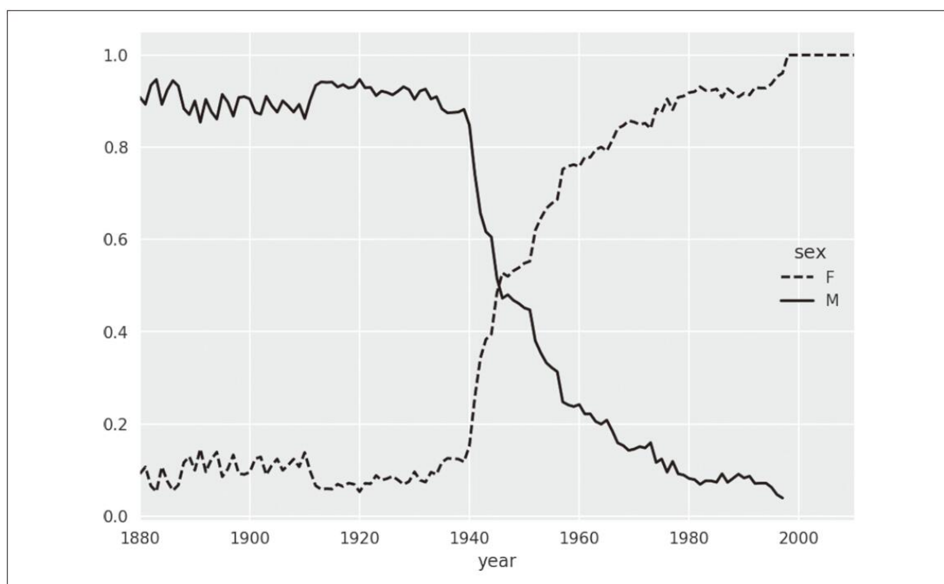


图14-10：随着时间推移男性/女性中Lesley式名字的比例

每种食物都有一些识别属性以及两份营养元素和营养比例的列表。这种形式的数据不适合分析，所以我们需要做一些工作来将数据转换成更好的形式。

从链接下载并提取数据后，你可以使用你选择的任何JSON库将其加载到Python中。我将使用内置的Python json模块：

```
In [154]: import json

In [155]: db = json.load(open('datasets/usda_food/database.json'))

In [156]: len(db)
Out[156]: 6636
```

db中的每个条目都是一个包含单个食物所有数据的词典。'nutrients'字段是一个字典的列表，每个营养元素对应一个字典：

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'gr
ons', 'nutrients'])

In [158]: db[0]['nutrients'][0]
```

```
Out[158]:
{'description': 'Protein',
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}

In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])

In [160]: nutrients[:7]
Out[160]:
                   description        group units    value
0                      Protein  Composition     g    25.18
1            Total lipid (fat)  Composition     g    29.20
2  Carbohydrate, by difference  Composition     g     3.06
3                          Ash        Other     g     3.28
4                       Energy       Energy  kcal   376.00
5                        Water  Composition     g    39.28
6                       Energy       Energy    kJ  1573.00
```

将字典的列表转换为DataFrame时，我们可以指定一个需要提取的字段列表。我们将提取食物名称、分类、ID和制造商：

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [162]: info = pd.DataFrame(db, columns=info_keys)

In [163]: info[:5]
Out[163]:
                          description                   group    id
0                     Cheese, caraway  Dairy and Egg Products  1008
1                     Cheese, cheddar  Dairy and Egg Products  1009
2                        Cheese, edam  Dairy and Egg Products  1018
3                        Cheese, feta  Dairy and Egg Products  1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products  1028
  manufacturer
0
1
2
3
4

In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description     6636 non-null object
group           6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

你可以通过value_counts查看食物组的分布情况：

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Fast Foods                           365
Legumes and Legume Products          365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
Name: group, dtype: int64
```

现在，对所有营养元素数据进行一些分析，将每种食物的营养元素组装成一张大表是最容易的。为此，我们需要采取几个步骤。首先，我会将食物营养元素的每个列表转换为DataFrame，为食物添加一列id，然后将DataFrame附加到列表中。然后，这些DataFrame可以通过concat连接在一起：

如果一切顺利，nutrients将如下：

```
In [167]: nutrients
Out[167]:
                               description        group units    valu
0                                  Protein  Composition     g   25.1
1                        Total lipid (fat)  Composition     g   29.2
2              Carbohydrate, by difference  Composition     g    3.0
3                                      Ash        Other     g    3.2
4                                   Energy       Energy  kcal  376.0
...                                    ...          ...   ...      .
389350                 Vitamin B-12, added     Vitamins   mcg    0.0
389351                         Cholesterol        Other    mg    0.0
389352         Fatty acids, total saturated        Other     g    0.0
389353  Fatty acids, total monounsaturated        Other     g    0.0
389354  Fatty acids, total polyunsaturated        Other     g    0.0
[389355 rows x 5 columns]
```

我注意到这个DataFrame中有重复的东西，所以删除重复值更好：

```
In [168]: nutrients.duplicated().sum()  # 重复的数量
Out[168]: 14179
```

```
In [169]: nutrients = nutrients.drop_duplicates()
```

因为'group'和'description'都是在DataFrame对象中的，我们可以明确地重命名：

```
In [170]: col_mapping = {'description' : 'food',
   .....:                'group'       : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, copy=False)

In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food            6636 non-null object
fgroup          6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
   .....:                'group' : 'nutgroup'}

In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=Fal

In [175]: nutrients
Out[175]:
                               nutrient     nutgroup units    valu
0                               Protein  Composition     g   25.1
1                     Total lipid (fat)  Composition     g   29.2
2           Carbohydrate, by difference  Composition     g    3.0
3                                   Ash        Other     g    3.2
4                                Energy       Energy  kcal  376.0
...                                 ...          ...   ...      .
389350              Vitamin B-12, added     Vitamins   mcg    0.0
389351                      Cholesterol        Other    mg    0.0
389352         Fatty acids, total saturated    Other     g    0.0
389353  Fatty acids, total monounsaturated     Other     g    0.0
389354  Fatty acids, total polyunsaturated     Other     g    0.0
[375176 rows x 5 columns]
```

完成所有这些工作后，我们准备将info与nutrients合并：

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')
In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
```

```
Data columns (total 8 columns):
nutrient        375176 non-null object
nutgroup        375176 non-null object
units           375176 non-null object
value           375176 non-null float64
id              375176 non-null int64
food            375176 non-null object
fgroup          375176 non-null object
manufacturer    293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB

In [178]: ndata.iloc[30000]
Out[178]:
nutrient                                       Glycine
nutgroup                                   Amino Acids
units                                                g
value                                             0.04
id                                                6158
food            Soup, tomato bisque, canned, condensed
fgroup                      Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

现在我们可以根据食物组和营养类型制作一个中位数图（见图14-11）：

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].qua
In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```

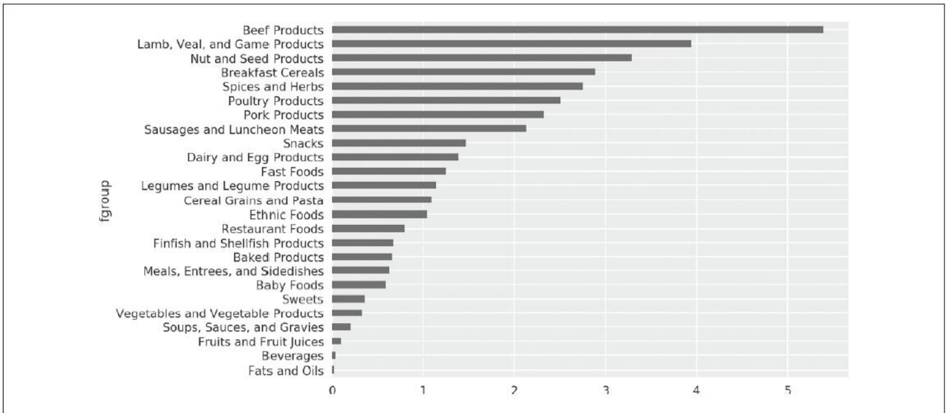


图14-11：按营养组的中位数锌值

只需要一点小聪明，你就可以发现哪种食物在每个营养元素下有最密集的营养：

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.loc[x.value.idxmax()]
get_minimum = lambda x: x.loc[x.value.idxmin()]

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# 使food小一点
max_foods.food = max_foods.food.str[:50]
```

由此产生的DataFrame有点太大而无法在书中显示。这里只有'Amino Acids'（氨基酸）营养组：

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine                          Gelatins, dry powder, unsweetened
Arginine                              Seeds, sesame flour, low-fat
Aspartic acid                                  Soy protein isolate
Cystine               Seeds, cottonseed flour, low fat (glandless)
Glutamic acid                                  Soy protein isolate
                                       ...
Serine           Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Threonine        Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Tryptophan        Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine         Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Valine           Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Name: food, Length: 19, dtype: object
```

14.5　2012年联邦选举委员会数据库

美国联邦选举委员会公布了有关政治运动贡献的数据。这些数据包括捐赠者姓名、职业和雇主、地址和缴费金额。一个有趣的数据来自2012年美国总统大选。我在2012年6月下载的一个数据集版本是一个150兆字节的CSV文件P00000001-ALL.csv（请参阅本书的数据存储库），该文件可以使用pandas.read_csv加载：

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')

In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id              1001731 non-null object
cand_id              1001731 non-null object
cand_nm              1001731 non-null object
contbr_nm            1001731 non-null object
contbr_city          1001712 non-null object
contbr_st            1001727 non-null object
contbr_zip           1001620 non-null object
contbr_employer       988002 non-null object
contbr_occupation     993301 non-null object
contb_receipt_amt    1001731 non-null float64
contb_receipt_dt     1001731 non-null object
receipt_desc           14166 non-null object
memo_cd                92482 non-null object
memo_text              97770 non-null object
form_tp              1001731 non-null object
file_num             1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

DataFrame中的一条样本记录如下：

```
In [186]: fec.iloc[123456]
Out[186]:
cmte_id                 C00431445
cand_id                 P80003338
cand_nm             Obama, Barack
contbr_nm             ELLMAN, IRA
contbr_city                 TEMPE
                        ...
receipt_desc                  NaN
memo_cd                       NaN
memo_text                     NaN
```

```
form_tp                    SA17A
file_num                  772372
Name: 123456, Length: 16, dtype: object
```

你可能会想到一些方法来切片和切块这些数据，以提取有关捐助者和竞选捐助模式的统计信息。我将向你展示一些应用了本书中技术的不同的分析。

你可以看到数据中没有政党背景，加入这些数据很有用。你可以使用unique获得所有不同的政治候选人名单：

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       "Roemer, Charles E. 'Buddy' III", 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, He
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

表示政党背景的方式之一是使用相应的字典 [1] ：

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',
           'Romney, Mitt': 'Republican',
           'Santorum, Rick': 'Republican'}
```

现在，在Series对象上使用map方法和上述的映射关系，你可以从候选人姓名中计算出政党的数组：

```
In [191]: fec.cand_nm[123456:123461]
```

```
Out[191]:
123456    Obama, Barack
123457    Obama, Barack
123458    Obama, Barack
123459    Obama, Barack
123460    Obama, Barack
Name: cand_nm, dtype: object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456    Democrat
123457    Democrat
123458    Democrat
123459    Democrat
123460    Democrat
Name: cand_nm, dtype: object

# 将它作为一列加入
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat      593746
Republican    407985
Name: party, dtype: int64
```

有一些数据准备的要点。首先，这些数据既包括捐款也包括退款（即负贡献金额）：

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True     991475
False     10256
Name: contb_receipt_amt, dtype: int64
```

为了简化分析，我将分析范围限制在正向贡献中：

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

由于Barack Obama和Mitt Romney是主要的两位候选人，我还将准备一个仅对他们的竞选有贡献的子集：

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney,
```

[1] 这里我们假设Gary Johnson是一名共和党人，尽管他后来加入了民主党。

14.5.1 按职业和雇主的捐献统计

根据职业分析捐献是一个常见的统计分析。例如，律师（法律代理人）倾向于捐更多的钱给民主党，而商务人士则倾向于捐更多的钱给共和党。你没有任何理由相信我，你可以自己看数据。首先，获得按职业的捐献总数是很简单的：

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                                   233990
INFORMATION REQUESTED                      35107
ATTORNEY                                   34286
HOMEMAKER                                  29931
PHYSICIAN                                  23432
INFORMATION REQUESTED PER BEST EFFORTS     21138
ENGINEER                                   14334
TEACHER                                    13990
CONSULTANT                                 13273
PROFESSOR                                  12555
Name: contbr_occupation, dtype: int64
```

通过查看职业，你会注意到很多捐款人都有相同的基础工作类型，或者说对于同一件事情有多个变量。下面的代码段表明如何通过将一种工作匹配到另一种来清理工作种类；请注意，使用dict.get存在一种"陷阱"，允许没有映射的职业"通过"：

```
occ_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
   'C.E.O.': 'CEO'
}

# 如果没有映射，则返回x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
我会对雇主字段做同样的事：
emp_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'SELF' : 'SELF-EMPLOYED',
   'SELF EMPLOYED' : 'SELF-EMPLOYED',
}
# 如果没有映射，则返回x
f = lambda x: emp_mapping.get(x, x)
```

```
fec.contbr_employer = fec.contbr_employer.map(f)
```

现在，你可以使用pivot_table按党派和职业聚合数据，然后过滤出至少捐赠200万美元的子集：

```
In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
   .....:                                 index='contbr_occupation',
   .....:                                 columns='party', aggfunc='sum')

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:
party                 Democrat    Republican
contbr_occupation
ATTORNEY           11141982.97  7.477194e+06
CEO                 2074974.79  4.211041e+06
CONSULTANT          2459912.71  2.544725e+06
ENGINEER             951525.55  1.818374e+06
EXECUTIVE           1355161.05  4.138850e+06
...                        ...           ...
PRESIDENT           1878509.95  4.720924e+06
PROFESSOR           2165071.08  2.967027e+05
REAL ESTATE          528902.09  1.625902e+06
RETIRED            25305116.38  2.356124e+07
SELF-EMPLOYED        672393.40  1.640253e+06
[17 rows x 2 columns]
```

以条形图的方式进行数据可视化更为简单（'barh'表示水平条形图，参见图14-12）：

```
In [205]: over_2mm.plot(kind='barh')
```

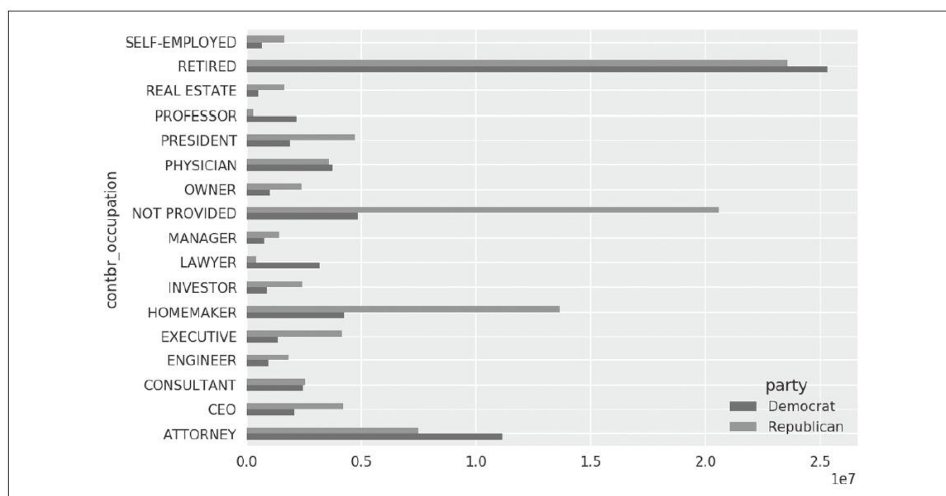


图14-12：按党派划分各职业捐赠总量

你可能对捐赠给Obama和Romney的顶级捐赠者的职业或顶级公司感兴趣。要实现这一点，你可以按候选人名称进行分组，并使用本章前面的top方法的变体：

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].sum()
    return totals.nlargest(n)
```

之后按职业和雇主进行聚合：

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[208]:
cand_nm        contbr_occupation
Obama, Barack  RETIRED                    25305116.38
               ATTORNEY                   11141982.97
               INFORMATION REQUESTED       4866973.96
               HOMEMAKER                   4248875.80
               PHYSICIAN                   3735124.94
                                              ...
Romney, Mitt   HOMEMAKER                   8147446.22
               ATTORNEY                    5364718.82
               PRESIDENT                   2491244.89
               EXECUTIVE                   2300947.03
               C.E.O.                      1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64
```

```
In [209]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[209]:
cand_nm        contbr_employer
Obama, Barack  RETIRED                   22694358.85
               SELF-EMPLOYED             17080985.96
               NOT EMPLOYED               8586308.70
               INFORMATION REQUESTED      5053480.37
               HOMEMAKER                  2605408.54
                                             ...
Romney, Mitt   CREDIT SUISSE               281150.00
               MORGAN STANLEY              267266.00
               GOLDMAN SACH & CO.          238250.00
               BARCLAYS CAPITAL            162750.00
               H.I.G. CAPITAL              139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

14.5.2　捐赠金额分桶

分析这些数据的一个有用的方法是使用cut函数将贡献者的数量按贡献大小离散化分桶：

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000,
   .....:                  100000, 1000000, 10000000])

In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [212]: labels
Out[212]:
411         (10, 100]
412       (100, 1000]
413       (100, 1000]
414         (10, 100]
415         (10, 100]
             ...
701381      (10, 100]
701382    (100, 1000]
701383        (1, 10]
701384      (10, 100]
701385    (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100] < (10
000, 10000] <
                               (10000, 100000] < (100000, 1000000] <
 10000000]]
```

然后，我们可以将Obama和Romney的数据按名称和分类标签进行分组，以获得捐赠规模的直方图：

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm              Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                       493.0          77.0
(1, 10]                    40070.0        3681.0
(10, 100]                 372280.0       31853.0
(100, 1000]               153991.0       43357.0
(1000, 10000]              22284.0       26186.0
(10000, 100000]                2.0           1.0
(100000, 1000000]              3.0           NaN
(1000000, 10000000]            4.0           NaN
```

这些数据表明，Obama获得的捐款数量比Romney大得多。你还可以对捐款数额进行求和并在桶内进行归一化，以便对按候选人划分的捐款总额百分比进行可视化（图14-13显示了结果图）：

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis

In [218]: normed_sums
Out[218]:
cand_nm              Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                    0.805182      0.194818
(1, 10]                   0.918767      0.081233
(10, 100]                 0.910769      0.089231
(100, 1000]               0.710176      0.289824
(1000, 10000]             0.447326      0.552674
(10000, 100000]           0.823120      0.176880
(100000, 1000000]         1.000000           NaN
(1000000, 10000000]       1.000000           NaN

In [219]: normed_sums[:-2].plot(kind='barh')
```

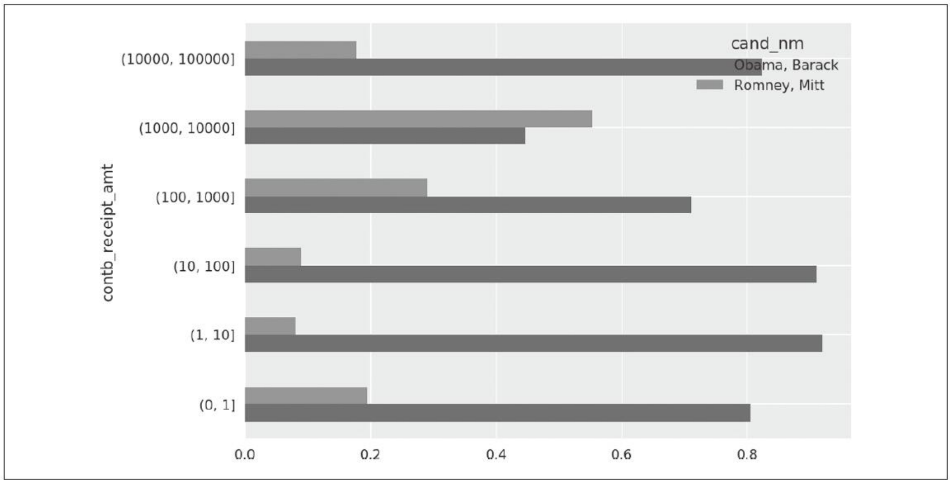


图14-13：不同捐赠规模的候选人收到的捐赠总额的百分比

我排除了两个最大的箱体，因为这些箱体不是由个人捐赠的。

这种分析可以通过多种方式进行改进和提高。例如，你可以通过捐助者姓名和邮政编码聚合捐款，以便为那些进行很多次小额捐赠的人进行调

整，他们并不会进行大型捐赠。我鼓励你自己下载并探索数据集。

14.5.3 按州进行捐赠统计

将数据按照候选人和州进行聚合是一项常规分析：

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK              281840.15      86204.24
AL              543123.48     527303.51
AR              359247.28     105556.00
AZ             1506476.98    1888436.23
CA            23824984.24   11237636.60
CO             2132429.49    1506714.12
CT             2068291.26    3499475.45
DC             4373538.80    1025137.50
DE              336669.14      82712.00
FL             7318178.58    8338458.81
```

如果你将每一行除以捐款总额，你就可以得到每个候选人按州的捐赠总额的相对百分比：

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK               0.765778      0.234222
AL               0.507390      0.492610
AR               0.772902      0.227098
AZ               0.443745      0.556255
CA               0.679498      0.320502
CO               0.585970      0.414030
CT               0.371476      0.628524
DC               0.810113      0.189887
DE               0.802776      0.197224
FL               0.467417      0.532583
```

14.6　本章小结

我们已经到达本书主要章节的尾声。我在附录中添加了一些额外的内容，这些内容可能会对你有用。

自从本书的第1版面世，这五年中Python已经成为一门广泛流行的数据分析语言。你在本书中学习到的编程技术将在未来相当长的时间内有效。我希望我们在本书中探索过的编程工具和类库可以在你的工作中发挥良好的作用。

附录A　高阶NumPy

在本附录中，我将更深入地介绍NumPy库的矩阵计算。内容将包括更多的ndarray内部细节和更多高级数组操作和算法。

本附录包含多个主题，无须按顺序阅读。

A.1　ndarray对象内幕

NumPy的ndarray提供了一种方法将一组同构数据（连续的或跨步的）解释为多维数组对象。数据类型或dtype决定数据如何被解释为浮点数、整数、布尔值或我们正在查看的任何其他类型。

让ndarray如此灵活的部分原因是每个数组对象都是一个数据块的分步视图。例如，你可能会想知道数组视图arr[：：2，：：-1]如何做到不复制任何数据。原因是ndarray不仅仅是一块内存和一个dtype，它还具有“跨步”信息，使数组能够以不同的步长在内存中移动。更准确地说，ndarray内部包含以下内容：

·指向数据的指针——即RAM中或内存映射文件中的数据块

·数据类型或dtype，描述数组中固定大小的值单元格

·表示数组形状（shape）的元组

·步长元组，表示要“步进”的字节数的整数以便沿维度推进一个元素

图A-1是简单的ndarray内部构造。

例如，一个10×5的数组，其shape为（10，5）：

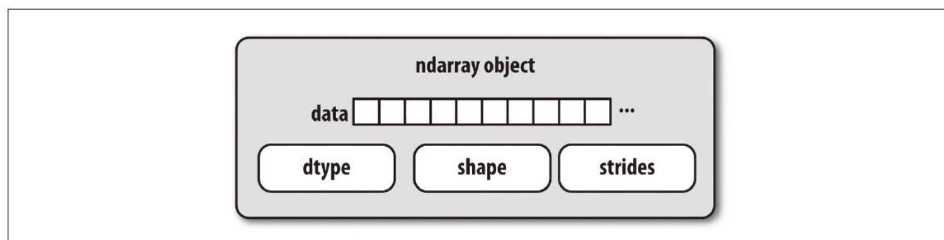
```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

一个典型的（C阶）3×4×5float64值（8字节）的数组具有跨度（160，40，8）（了解跨度可能是有用的，因为通常特定轴上的跨度越大，沿着该轴执行计算的代价越高）：

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
```

```
Out[11]: (160, 40, 8)
```

虽然一般的NumPy用户很少会对数组跨度（strides）感兴趣，但它们是构建“零复制”数组视图的关键因素。跨度甚至可以是负的，这使得数组能够穿过内存“向后”移动（例如，在诸如obj[：：-1]或obj[：，：：-1]的切片中就是这种情况）。



图A-1：NumPy的ndarray对象

A.1.1 NumPy dtype层次结构

你可能时不时会需要写代码检查数组是否包含整数、浮点数、字符串或Python对象。由于浮点数有多种类型（float16到float128），因此检查dtype是否在类型列表中会非常麻烦。幸运的是，dtype有超类，如np.integer和np.floating，它们可以和np.issubdtype函数一起使用：

```
In [12]: ints = np.ones(10, dtype=np.uint16)

In [13]: floats = np.ones(10, dtype=np.float32)

In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True

In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

你可以通过调用类型的mro方法来查看特定dtype的所有父类：

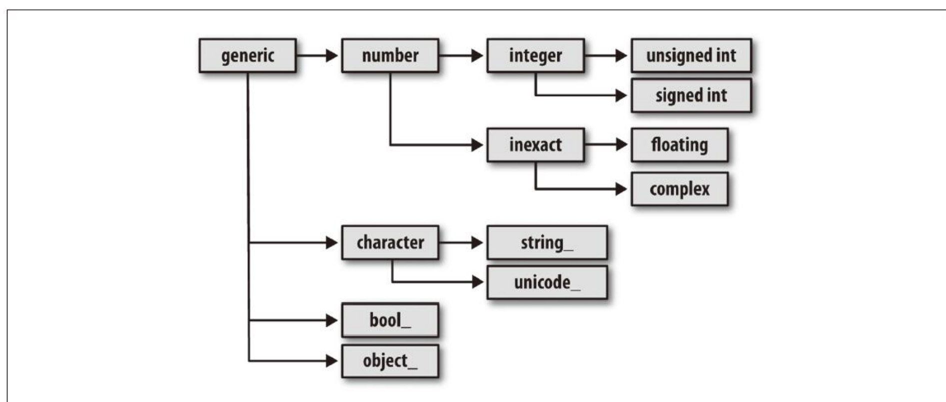
```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
```

```
    numpy.generic,
    float,
    object]
```

因此，我们可以得到：

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

大部分NumPy用户不必知道这点，但知道了就偶尔可以派上用场。图A-2是dtype的层次结构父类-子类关系图。[1]



图A-2：NumPy dtype类型层级关系

A.2 高阶数组操作

除了神奇索引、切片和布尔值子集外，还有很多方式可以处理数组。虽然大部分数据分析应用程序的繁重工作都是由pandas中的高级函数处理的，但有时候你可能需要编写一些在现有库中找不到的数据算法。

A.2.1 重塑数组

在很多情况下，你将数组从一个形状转换为另一个形状，并且不复制任何数据。为了实现这个功能，可以向reshape数组实例方法传递一个表示新形状的元组。例如，假设我们有一个一维数组，我们想要把该数组重新排列进一个矩阵（结果展示在图A-3中）：

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

图A-3：按C顺序（行方向）的重塑和按Fortran顺序（列方向）的重塑

多维数组也可以被重塑：

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

传递的形状维度可以有一个值是-1，表示维度通过数据进行推断：

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
```

```
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

由于数组的shape属性是一个元组，它也可以被传递给reshape：

```
In [24]: other_arr = np.ones((3, 5))

In [25]: other_arr.shape
Out[25]: (3, 5)

In [26]: arr.reshape(other_arr.shape)
Out[26]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

reshape的反操作可以将更高维度的数组转换为一维数组，这种操作通常被成为扁平化（flattening）或分散化（raveling）：

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [29]: arr.ravel()
Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 1
```

如果结果中的值在原始数组中是连续的，则ravel不会生成底层数值的副本。flatten方法的行为类似于ravel，但它总是返回数据的副本：

```
In [30]: arr.flatten()
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 1
```

数据可以按照不同的顺序进行重塑或扁平化。顺序对于新的NumPy用户来说，这是一个有点微妙的主题，因此是下一个子主题。

A.2.2　C顺序和Fortran顺序

NumPy为你提供内存中数据布局的控制和灵活性。默认情况下，NumPy数组是按行方向顺序创建的。在空间上，这意味着如果你有一个二维的数据数组，数组每行中的元素存储在相邻的存储单元中。行方向顺序的替代方法是列方向顺序，这意味着每列数据中的值都存储在相邻的内存位置中。

由于历史原因，行和列方向的顺序也分别称为C顺序和Fortran顺序。在FORTRAN 77语言中，矩阵都是列方向的。

像reshape和ravel函数接收一个order参数，该参数表示数据在数组中使用哪种顺序。在大部分情况下，该参数可以被设置为'C'或'F'（还有一些不太常用的选项'A'和'K'。请参考NumPy官方文档，然后回看图A-3对这些选项的表示）。

```
In [31]: arr = np.arange(12).reshape((3, 4))

In [32]: arr
Out[32]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [33]: arr.ravel()
Out[33]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [34]: arr.ravel('F')
Out[34]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

对二维以上的数组重塑可能会有点令人费解（见图A-3）。C顺序和Fortran顺序的核心区别就是在维度方向上遍历的方式：

C顺序/行方向顺序

首先遍历更高的维度（例如，在轴0上行进之前先在轴1上行进）

Fortran顺序/列方向顺序

最后遍历更高的维度（例如，在轴1上行进之前先在轴0上行进）

A.2.3　连接和分隔数组

numpy.concatenate可以获取数组的序列（元组、列表等），并沿着输入轴将它们按顺序连接在一起：

```
In [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [37]: np.concatenate([arr1, arr2], axis=0)
Out[37]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [38]: np.concatenate([arr1, arr2], axis=1)
Out[38]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

对于常见的连接类型有一些方便的函数，比如vstack和hstack。之前的操作可以这样表达：

```
In [39]: np.vstack((arr1, arr2))
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [40]: np.hstack((arr1, arr2))
Out[40]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

另一方面，split可以将一个数组沿轴向切片成多个数组：

```
In [41]: arr = np.random.randn(5, 2)

In [42]: arr
Out[42]:
array([[-0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])

In [43]: first, second, third = np.split(arr, [1, 3])
```

```
In [44]: first
Out[44]: array([[-0.2047,  0.4789]])

In [45]: second
Out[45]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])

In [46]: third
Out[46]:
array([[ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

传递给np.split的值[1，3]表示将数组拆分时的索引位置。

表A-1是全部有关连接和分隔的函数列表，其中一些仅作为通用concatenate的便捷方法。

表A-1：数组连接函数

函数	描述
concatenate	最通用的函数，沿一个轴向连接数组的集合
vstack, row_stack	按行堆叠数组（沿着轴 0）
hstack	按列堆叠数组（沿着轴 1）
column_stack	类似于 hstack，但会首先把 1 维数组转换为 2 维列向量
dstack	按“深度”堆叠数组（沿着轴 2）
split	沿着指定的轴，在传递的位置上分隔数组
hsplit/vsplit	分别是沿着轴 0 和轴 1 进行分隔的方便函数

A.2.3.1 堆叠助手：r_和c_

在NumPy命名空间中有两个特殊的对象：r_和c_，它们可以使堆栈数组的操作更为简洁：

```
In [47]: arr = np.arange(6)

In [48]: arr1 = arr.reshape((3, 2))

In [49]: arr2 = np.random.randn(3, 2)

In [50]: np.r_[arr1, arr2]
Out[50]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
```

```
       [ 4.    ,  5.    ],
       [ 1.0072, -1.2962],
       [ 0.275 ,  0.2289],
       [ 1.3529,  0.8864]])

In [51]: np.c_[np.r_[arr1, arr2], arr]
Out[51]:
array([[ 0.    ,  1.    ,  0.    ],
       [ 2.    ,  3.    ,  1.    ],
       [ 4.    ,  5.    ,  2.    ],
       [ 1.0072, -1.2962,  3.    ],
       [ 0.275 ,  0.2289,  4.    ],
       [ 1.3529,  0.8864,  5.    ]])
```

这些函数还可以将切片转换为数组：

```
In [52]: np.c_[1:6, -10:-5]
Out[52]:
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

参看官方文档可以让你了解到更多c_和r_的功能。

A.2.4 重复元素：tile和repeat

repeat和tile函数是用于重复或复制数组的两个有用的工具。repeat函数按照给定次数对数组中的每个元素进行复制，生成一个更大的数组：

```
In [53]: arr = np.arange(3)

In [54]: arr
Out[54]: array([0, 1, 2])

In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

对于NumPy而言，复制或重复数组的需求可能不如其他数组编程框架（如MATLAB）那样常见。其中一个原因是广播通常会更好地满足这一需求，这是下一部分的主题。

默认情况下，如果你传递一个整数，每个元素都会复制相应的次数。如果你传递了一个整数数组，每个元素都会重复相应的不同次数：

```
In [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

多维数组可以在指定的轴向上对它们的元素进行重复：

```
In [57]: arr = np.random.randn(2, 2)

In [58]: arr
Out[58]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [59]: arr.repeat(2, axis=0)
Out[59]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

请注意，如果没有传递轴，数组将首先扁平化，这可能不是你想要的。同样，需要按照不同次数重复多维数组的切片时，你可以传递一个整数数组：

```
In [60]: arr.repeat([2, 3], axis=0)
Out[60]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])

In [61]: arr.repeat([2, 3], axis=1)
Out[61]:
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]])
```

另一方面，tile是一种快捷方法，它可以沿着轴向堆叠副本。在视觉上，你可以把它看作类似于“铺设瓷砖”：

```
In [62]: arr
```

```
Out[62]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])
In [63]: np.tile(arr, 2)
Out[63]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

第二个参数是瓷砖的数量。用标量来说，铺设是逐行进行的，而不是逐列。tile的第二个参数可以是表示“铺瓷砖”布局的元组：

```
In [64]: arr
Out[64]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [65]: np.tile(arr, (2, 1))
Out[65]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [66]: np.tile(arr, (3, 2))
Out[66]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

A.2.5 神奇索引的等价方法：take和put

如果你回忆第4章，使用整数数组通过神奇索引是获取、设置数组子集的一种方式：

```
In [67]: arr = np.arange(10) * 100

In [68]: inds = [7, 1, 2, 6]

In [69]: arr[inds]
Out[69]: array([700, 100, 200, 600])
```

还有其他一些ndarray方法可以用于特殊情况下在单个轴上的数据选

取：

```
In [70]: arr.take(inds)
Out[70]: array([700, 100, 200, 600])

In [71]: arr.put(inds, 42)

In [72]: arr
Out[72]: array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])

In [73]: arr.put(inds, [40, 41, 42, 43])

In [74]: arr
Out[74]: array([  0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

如果要在别的轴上使用take，你可以传递axis关键字：

```
In [75]: inds = [2, 0, 2, 1]

In [76]: arr = np.random.randn(2, 4)

In [77]: arr
Out[77]:
array([[-0.5397,  0.477 ,  3.2489, -1.0212],
       [-0.5771,  0.1241,  0.3026,  0.5238]])

In [78]: arr.take(inds, axis=1)
Out[78]:
array([[ 3.2489, -0.5397,  3.2489,  0.477 ],
       [ 0.3026, -0.5771,  0.3026,  0.1241]])
```

put不接受axis参数，而是将数组索引到扁平版本（一维，C顺序）。因此，当您需要使用其他轴上的索引数组设置元素时，通常最容易使用神奇索引。

A.3 广播

广播描述了算法如何在不同形状的数组之间进行运算。它是一个强大的功能，但可能会导致混淆，即使对于有经验的用户也是如此。最简单的广播示例发生在将标量值与数组组合的时候：

```
In [79]: arr = np.arange(5)

In [80]: arr
```

```
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: array([ 0,  4,  8, 12, 16])
```

这里我们说标量值4已经被广播给乘法运算中的所有其他元素。

例如，我们可以通过减去列均值来降低数组中的每一列的数值。在这种情况下，它非常简单：

```
In [82]: arr = np.random.randn(4, 3)

In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])

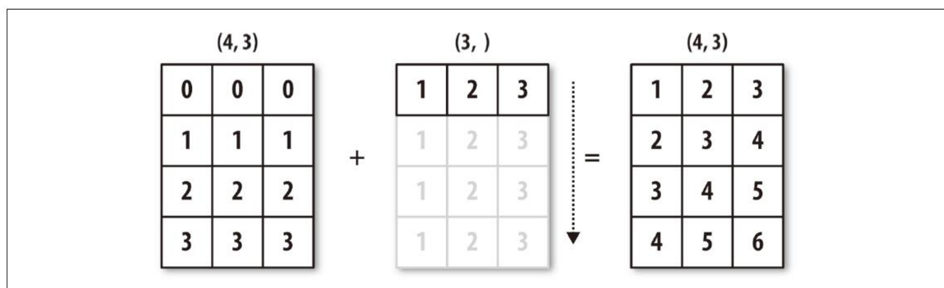
In [86]: demeaned.mean(0)
Out[86]: array([-0.,  0., -0.])
```

有关此操作的说明，请参见图A-4。对行进行减均值的广播需要更小心。幸运的是，只要遵循规则，就可以在数组的任何维度上对潜在较低维度值进行广播（例如从二维数组的每一列中减去行均值）。

于是我们有：

广播的规则

如果对于每个结尾维度（即从尾部开始的），轴长度都匹配或者长度都是1，两个二维数组就是可以兼容广播的。之后，广播会在丢失的或长度为1的轴上进行。



图A-4：一维数组沿轴0进行广播

尽管我也是个经验丰富的NumPy用户，但我也不得不经常在思考广播机制时，停下来画一张图。考虑上一个例子，假设我们希望减去每一行的平均值。由于arr.mean（0）的长度为3，因此它与轴0上的广播兼容，因为arr中的结尾维度为3，因此匹配。

根据规则，为了从轴1减均值（即从每行减去行平均值），较小的数组的形状必须是（4，1）：

```
In [87]: arr
Out[87]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])
In [88]: row_means = arr.mean(1)

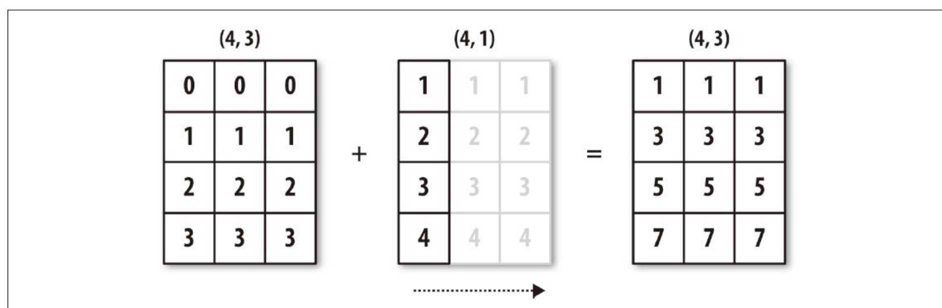
In [89]: row_means.shape
Out[89]: (4,)

In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

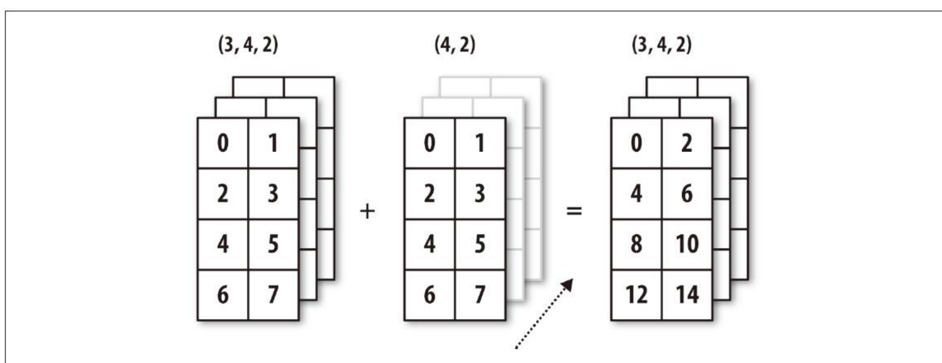
In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0.,  0.,  0.])
```

图A-5阐明了该操作。



图A-5：沿着轴1对2维数组进行广播

图A-6是对沿着轴0将一个二维数组加到三维数组的示意。



图A-6：沿着轴0对三维数组的广播

A.3.1　在其他轴上广播

使用更高维度的数组进行广播可能会更加令人头痛，但这确实是遵循规则的问题。如果你不这样做，你会得到这样的错误：

```
In [93]: arr - arr.mean(1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call la
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,
```

想要在轴0以外的轴上使用较低维数组进行算术运算是相当普遍的。

根据广播规则，“广播维度”在较小的数组中必须为1。在这里显示的行减均值的例子中，这表示重新塑造行意味着形状是（4，1）而不是（4，）：

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095,  1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

在三维情况下，在三个维度中的任何一个维度上进行广播只是将数据重塑为形状兼容的问题。图A-7很好地显示了在三维数组的每个轴上广播所需的形状。

因此，一个常见的问题是需要添加一个长度为1的新轴，专门用于广播目的。使用reshape是一种选择，但插入一个轴需要构造一个表示新形状的元组。这通常是一个乏味的练习。因此，NumPy数组提供了一种通过索引插入新轴的特殊语法。我们使用特殊的np.newaxis属性和“完整”切片来插入新轴：

```
In [95]: arr = np.zeros((4, 4))

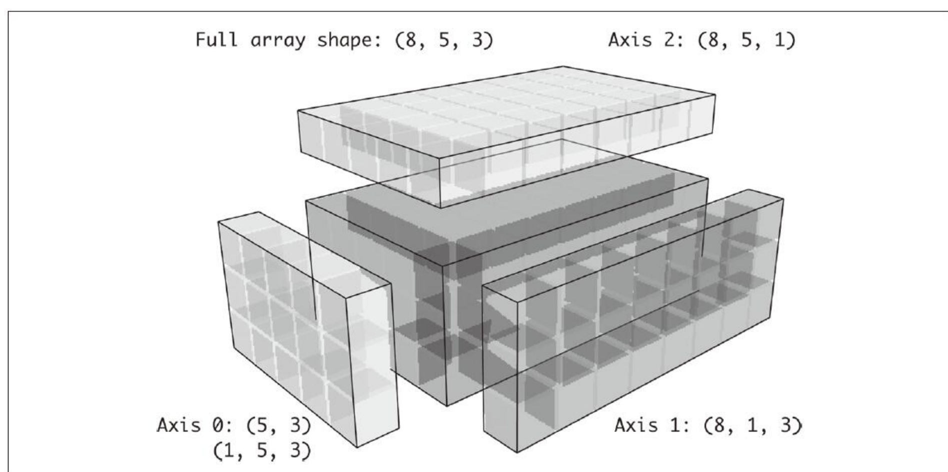
In [96]: arr_3d = arr[:, np.newaxis, :]

In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]:
array([[-2.3594],
       [-0.1995],
       [-1.542 ]])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542 ]])
```



图A-7：用于在三维数组上广播的兼容的二维数组形状

因此，如果我有一个三维数组并想在轴2上减去均值，也就是说我们需要这样写：

```
In [101]: arr = np.random.randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newaxis]

In [106]: demeaned.mean(2)
Out[106]:
array([[ 0.,  0., -0., -0.],
       [ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

你可能会想是否有一个在轴向上减均值但又不牺牲性能的方法。有的，但是它需要一些索引操作：

```
def demean_axis(arr, axis=0):
```

```
    means = arr.mean(axis)
    # 将形如[:, :, np.newaxis]的对象推广到N维
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

A.3.2 通过广播设定数组的值

控制算术运算的相同广播规则也适用于通过数组索引设置值。在一个简单的例子中，我们可以做这样的事情：

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr
Out[109]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

但是，如果我们想要将数值的一维数组设置到数组的列中，只要形状是兼容的就可以做到这一点：

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6  ]])
```

A.4 高阶ufunc用法

虽然许多NumPy用户只会使用通用函数提供的按元素操作，但还有一些额外的功能偶尔可以帮助你编写更简洁的代码而无须循环。

A.4.1 ufunc实例方法

NumPy的每个二元ufunc（通用函数）都有特殊的方法来执行某些特殊的向量化操作。表A-2总结了这些内容，但我将举几个具体示例来说明它们的工作原理。

reduce方法接收单个数组并通过执行一系列二元操作在可选的轴向上对数组的值进行聚合。例如，使用np.add.reduce是对数组中元素进行加和的另一种方法：

```
In [115]: arr = np.arange(10)

In [116]: np.add.reduce(arr)
Out[116]: 45

In [117]: arr.sum()
Out[117]: 45
```

起始值（对于add方法是0）取决于ufunc。如果传递了一个轴，则沿该轴执行缩聚。这使你能够以简洁的方式回答某些种类的问题。作为一个不太重要的例子，我们可以使用np.logical_and来检查数组的每一行中的值是否被排序：

```
In [118]: np.random.seed(12346)   # 为了可以复现

In [119]: arr = np.random.randn(5, 5)

In [120]: arr[::2].sort(1) # 对行排序

In [121]: arr[:, :-1] < arr[:, 1:]
Out[121]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
```

```
Out[122]: array([ True, False,  True, False,  True], dtype=bool)
```

请注意，logical_and.reduce等价于all方法。

accumulate与reduce是相关的，就像cumsum与sum相关一样。accumulate生成一个数组，其尺寸与中间“累计”值相同：

```
In [123]: arr = np.arange(15).reshape((3, 5))

In [124]: np.add.accumulate(arr, axis=1)
Out[124]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

outer在两个数组之间执行成对的交叉乘积：

```
In [125]: arr = np.arange(3).repeat([1, 2, 2])

In [126]: arr
Out[126]: array([0, 1, 1, 2, 2])
In [127]: np.multiply.outer(arr, np.arange(5))
Out[127]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

outer的输出的维度等于输入的维度总和：

```
In [128]: x, y = np.random.randn(3, 4), np.random.randn(5)

In [129]: result = np.subtract.outer(x, y)

In [130]: result.shape
Out[130]: (3, 4, 5)
```

最后一个方法reduceat执行“本地缩聚”，本质上是一个数组groupby操作，在操作中数组的切片聚合在了一起。reduceat方法接受一系列的“箱体边缘”，这些箱体边缘表示如何分隔以及聚合数据值：

```
In [131]: arr = np.arange(10)

In [132]: np.add.reduceat(arr, [0, 5, 8])
Out[132]: array([10, 18, 17])
```

结果是在arr[0：5]、arr[5：8]和arr[8：]上执行了缩聚（此处是加和）。在其他方法中，你可以传递一个axis参数：

```
In [133]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [134]: arr
Out[134]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

表A-2列出了部分ufunc方法。

表A-2：ufunc方法

方法	描述
reduce (x)	按操作的连续应用程序对数值聚合
accumulate (x)	聚合值，保留所有部分聚合
reduceat (x, bins)	“本地”缩聚或“group by”，减少连续的数据切片以生成聚合数组

表A-2：ufunc方法（续）

方法	描述
outer (x, y)	将操作应用于 x 和 y 中的所有元素对，结果数组的形状为 x.shape + y.shape

A.4.2 使用Python编写新的ufunc方法

有很多工具可以用于创建你自己的NumPy ufunc，最常用的是NumPy

的C语言API，但是它已经超出了本书的范畴。在本节，我们将一起看看纯Python的ufunc方法。

numpy.frompyfunc函数接收一个具有特定数字输入和输出的函数。例如，一个简单的按元素相加的函数可以如下：

```
In [136]: def add_elements(x, y):
   .....:     return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1)

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

使用frompyfunc创建的函数通常返回的是Python对象的数组，这并不方便。幸运的是，还有另一个函数numpy.vectorize允许指定输出的类型（但功能稍差）：

```
In [139]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.])
```

这些函数提供了一种创建类似ufunc的函数的方法，但是它们非常慢，因为它们需要Python函数调用来计算每个元素，这比NumPy的基于C的ufunc循环要慢很多：

```
In [141]: arr = np.random.randn(10000)

In [142]: %timeit add_them(arr, arr)
4.12 ms +- 182 us per loop (mean +- std. dev. of 7 runs, 100 loops ea

In [143]: %timeit np.add(arr, arr)
6.89 us +- 504 ns per loop (mean +- std. dev. of 7 runs, 100000 loop
```

本章的后续内容会介绍如何使用Numba库（http://numba.pydata.org/ ）用Python语言创建快速ufunc。

A.5 结构化和记录数组

到目前为止，你可能已经注意到，ndarray是一个同构数据的容器。

也就是说，它表示一个内存块，其中每个元素占用相同数量的字节，由dtype确定。表面上，ndarray的这种特性不允许你使用它表示异构的数据或表格型数据。结构化数组是一个ndarray，其中每个元素可以被认为代表C中的struct（因此是“结构化”的名称），或者是SQL表中具有多个命名字段的行：

```
In [144]: dtype = [('x', np.float64), ('y', np.int32)]

In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

In [146]: sarr
Out[146]:
array([( 1.5   ,  6), ( 3.1416, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

有几种方法可以指定结构化的dtype（请参阅NumPy官方在线文档）。一种典型的方式是使用（field_name，field_data_type）作为元组的列表。现在，数组的元素是元组对象，其元素可以像字典一样访问：

```
In [147]: sarr[0]
Out[147]: ( 1.5, 6)

In [148]: sarr[0]['y']
Out[148]: 6
```

字段名称存储在dtype.names属性中。当你访问结构化数组中的字段时，将返回数据的分步视图，因此不会复制任何内容：

```
In [149]: sarr['x']
Out[149]: array([ 1.5   ,  3.1416])
```

A.5.1　嵌套dtype和多维字段

当指定结构化的dtype时，你可以另外传递一个形状（以int或元组的形式）：

```
In [150]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [151]: arr = np.zeros(4, dtype=dtype)
```

```
In [152]: arr
Out[152]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

在这种情况下，x字段引用的是每条记录中长度为3的数组：

```
In [153]: arr[0]['x']
Out[153]: array([0, 0, 0])
```

很方便地，访问arr['x']然后返回一个二维数组，而不是像之前的例子那样返回一个一维数组：

```
In [154]: arr['x']
Out[154]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

这使你可以将更复杂的嵌套结构表示为数组中的单个内存块。你也可以嵌套dtype来创建更复杂的结构。这里是一个例子：

```
In [155]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32

In [156]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [157]: data['x']
Out[157]:
array([( 1.,  2.), ( 3.,  4.)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [158]: data['y']
Out[158]: array([5, 6], dtype=int32)

In [159]: data['x']['a']
Out[159]: array([ 1.,  3.])
```

pandas的DataFrame并不直接支持这个特性，尽管它与分层索引很相似。

A.5.2 为什么要使用结构化数组

与pandas的DataFrame相比，NumPy结构化数组是一个相对底层的工具。结构化数组提供了一种将内存块解释为具有任意复杂嵌套列的表格结构的方法。由于数组中的每个元素都在内存中表示为固定数量的字节，因此结构化数组提供了读/写磁盘（包括内存映射）数据，以及在网络上传输数据和其他此类用途的非常快速有效的方法。

作为结构化数组的另一种常见用途，将数据文件编写为固定长度的记录字节流是将C和C++代码中的数据序列化的常用方法，这在业界传统系统中很常见。只要知道文件的格式（每个记录的大小以及每个元素的顺序、字节大小和数据类型），就可以用np.fromfile将数据读入内存。像这样的专门用途超出了本书的范围，但值得知道的是这样的实现是可能的。

A.6　更多关于排序的内容

和Python的内建列表类似，ndarray的sort实例方法是一种原位排序，意味着数组的内容进行了重排列，而不是生成了一个新的数组：

```
In [160]: arr = np.random.randn(6)

In [161]: arr.sort()

In [162]: arr
Out[162]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,  1.841
```

在进行数组原位排序时，请记住如果数组是不同ndarray的视图的话，原始数组将会被改变：

```
In [163]: arr = np.random.randn(3, 5)

In [164]: arr
Out[164]:
array([[-0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])

In [165]: arr[:, 0].sort()  # 对第一列的值原位排序

In [166]: arr
Out[166]:
array([[-1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

另一方面，numpy.sort产生的是一个数组的新的、排序后的副本。否则，它接受与ndarray.sort相同的参数（如kind）：

```
In [167]: arr = np.random.randn(5)

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])

In [169]: np.sort(arr)
Out[169]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])

In [170]: arr
Out[170]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

所有这些排序方法都有一个axis参数，用于独立地沿着传递的轴对数据部分进行排序：

```
In [171]: arr = np.random.randn(3, 5)

In [172]: arr
Out[172]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])

In [173]: arr.sort(axis=1)

In [174]: arr
Out[174]:
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

你可能会注意到所有的排序方法都没有降序排列的选项。这是一个实践中的问题，因为数组切片会产生视图，因此不需要生成副本也不需要任何计算工作。很多Python用户对于列表（假设列表名为values）的一种"技巧"很熟悉，即values[：：-1]会返回一个反序的列表。对ndarray也是一样：

```
In [175]: arr[:, ::-1]
Out[175]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

A.6.1 间接排序：argsort和lexsort

在数据分析中，你可能需要通过一个或多个键对数据集进行重新排序。例如，有关某些学生的数据表可能需要按姓氏排序，然后按名字排序。这是一个间接排序的例子，如果你读过pandas相关的章节，你已经看到了许多更高级的例子。给定一个或多个键（一个或多个值的数组），你希望获得一个整数索引（我将它们通称为索引器）数组，整数索引将告诉你如何重新排列数据为指定顺序。两种实现该功能的方法是argsort和numpy.lexsort。举个例子：

```
In [176]: values = np.array([5, 0, 1, 3, 2])

In [177]: indexer = values.argsort()

In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])

In [179]: values[indexer]
Out[179]: array([0, 1, 2, 3, 5])
```

作为一个更复杂的例子，下面的代码对一个二维数组按照它的第一行进行重新排序：

```
In [180]: arr = np.random.randn(3, 5)

In [181]: arr[0] = values

In [182]: arr
Out[182]:
array([[ 5.    ,  0.    ,  1.    ,  3.    ,  2.    ],
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],
       [-0.2089,  0.2316,  0.728 , -1.3918,  1.9956]])

In [183]: arr[:, arr[0].argsort()]
Out[183]:
array([[ 0.    ,  1.    ,  2.    ,  3.    ,  5.    ],
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],
       [ 0.2316,  0.728 ,  1.9956, -1.3918, -0.2089]])
```

lexsort类似于argsort，但它对多键数组执行间接字典排序。假设我们想对一些由名字和姓氏标识的数据进行排序：

```
In [184]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Ba
```

```
In [185]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones',

In [186]: sorter = np.lexsort((first_name, last_name))

In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])

In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7fa203eda1c8>
```

在你第一次使用lexsort时，lexsort可能有点令人困惑，因为用于排序数据的键的顺序从传递的最后一个数组开始。这里，last_name在first_name之前使用。

pandas的方法，比如Series和DataFrame的sort_values方法是对这些方法的变相实现（这些方法也必须要考虑缺失值）。

A.6.2　其他的排序算法

稳定排序算法保留了相等元素的相对位置。在相对顺序有意义的间接排序中，这可能尤其重要：

```
In [189]: values = np.array(['2:first', '2:second', '1:first', '1:se
     .....:                    '1:third'])

In [190]: key = np.array([2, 2, 1, 1, 1])

In [191]: indexer = key.argsort(kind='mergesort')

In [192]: indexer
Out[192]: array([2, 3, 4, 0, 1])

In [193]: values.take(indexer)
Out[193]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

唯一可用的稳定排序是mergesort，它保证了O（n log n）性能（对于复杂性增益），但其平均性能比默认的quicksort方法更差。请参阅表A-3，了解可用方法及其相对性能（和性能保证）的总结。这是大多数用户永远不必考虑的事情，但知道它的存在是有用的。

表A-3：数组排序方法

种类	速度	是否稳定	工作空间	最差情况
'quicksort'	1	No	0	O (n^2)
'mergesort'	2	Yes	n / 2	O(n log n)
'heapsort'	3	No	0	O(n log n)

A.6.3 数组的部分排序

排序的目标之一可以是确定数组中最大或最小的元素。NumPy已经优化了方法numpy.partition和np.argpartition，用于围绕第k个最小元素对数组进行分区：

```
In [194]: np.random.seed(12345)

In [195]: arr = np.random.randn(20)

In [196]: arr
Out[196]:
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,  1.3934,  0.0929,
        0.2817,  0.769 ,  1.2464,  1.0072, -1.2962,  0.275 ,  0.2289,
        1.3529,  0.8864, -2.0016, -0.3718,  1.669 , -0.4386])

In [197]: np.partition(arr, 3)
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

在调用partition（arr，3）之后，结果中的前三个元素是最小的三个值，并不是特定的顺序。numpy.argpartition类似于numpy.argsort排序，它返回的是将数据重新排列为等价顺序的索引：

```
In [198]: indices = np.argpartition(arr, 3)

In [199]: indices
Out[199]:
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13, 14, 15,
        4, 18,  9])

In [200]: arr.take(indices)
Out[200]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

A.6.4 numpy.searchsorted：在已排序的数组寻找元素

searchsorted是一个数组方法，它对已排序数组执行二分搜索，返回数组中需要插入值的位置以保持排序：

```
In [201]: arr = np.array([0, 1, 7, 12, 15])

In [202]: arr.searchsorted(9)
Out[202]: 3
```

你还可以传递一个值数组来获取一个索引数组：

```
In [203]: arr.searchsorted([0, 8, 11, 16])
Out[203]: array([0, 3, 3, 5])
```

你可能已经注意到，对于0元素，searchsorted返回0。这是因为默认行为是返回一组相等值左侧的索引：

```
In [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [205]: arr.searchsorted([0, 1])
Out[205]: array([0, 3])

In [206]: arr.searchsorted([0, 1], side='right')
Out[206]: array([3, 7])
```

作为searchsorted的另一个应用，假设我们有一个介于0和10，000之间的数值，以及我们想用来分隔数据的单独的“桶边界”数组：

```
In [207]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [208]: bins = np.array([0, 100, 1000, 5000, 10000])

In [209]: data
Out[209]:
array([ 9940.,   6768.,   7908.,   1709.,    268.,   8003.,   9037.,    246
        4917.,   5262.,   5963.,    519.,   8950.,   7282.,   8183.,   5002
        8101.,    959.,   2189.,   2587.,   4681.,   4593.,   7095.,   1780
        5314.,   1677.,   7688.,   9281.,   6094.,   1501.,   4896.,   3773
        8486.,   9110.,   3838.,   3154.,   5683.,   1878.,   1258.,   6875
        7996.,   5735.,   9732.,   6340.,   8884.,   4954.,   3516.,   7142
        5039.,   2256.])
```

然后得到每个数据点属于哪个区间的标签（其中1代表桶[0，100）），我们可以简单地使用searchsorted：

```
In [210]: labels = bins.searchsorted(data)

In [211]: labels
Out[211]:
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 2, 3, 3, 3,
       3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3, 4, 4, 4, 4, 4,
       3, 4, 4, 3])
```

可以和pandas的groupby一起被用于分箱数据：

```
In [212]: pd.Series(data).groupby(labels).mean()
Out[212]:
2     498.000000
3    3064.277778
4    7389.035714
dtype: float64
```

A.7　使用Numba编写快速NumPy函数

Numba（http://numba.pydata.org ）是一个开源项目，可为使用CPU、GPU或其他硬件的NumPy类型的数据创建快速函数。它使用LLVM Project（http://llvm.org ）将Python代码翻译成编译后的机器码。

为了介绍Numba，让我们考虑一个纯Python函数，该函数使用for循环计算表达式（x-y）.mean（）的值：

```
import numpy as np
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

这个函数是非常慢的：

```
In [209]: x = np.random.randn(10000000)

In [210]: y = np.random.randn(10000000)

In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop

In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

NumPy的版本要快100倍。我们可以使用numba.jit函数将这个函数编译成Numba函数：

```
In [213]: import numba as nb

In [214]: numba_mean_distance = nb.jit(mean_distance)
```

我们也可以写成装饰器的形式：

```
@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

结果函数实际上比矢量化的NumPy版本更快：

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba不能编译所有的Python代码，但它支持纯Python代码的重要子集，这些代码对于编写数值算法最为有用。

Numba是一个有深度的类库，支持各种不同的硬件、兼容模式和用户拓展。它还可以在不显式使用for循环的情况下兼容NumPy的Python API。Numba能够识别可以编译为机器代码的构造，同时将不知道如何编译的函数的调用替换为CPython API。Numba的jit函数有一个选项，nopython=True，它将允许的代码限制为可以编译为LLVM的Python代

码，而无须调用任何Python的C语言API。jit（nopython=True）有一个更短的别名numba.njit。

在之前的例子中，我们可以写为：

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

我推荐你通过阅读Numba的官方在线文档（http://numba.pydata.org）来学习更多内容。下一节将会展示一个创建自定义NumPy ufunc对象的例子。

A.7.1 使用Numba创建自定义numpy.ufunc对象

numba.vectorize函数创建了编译好的NumPy ufunc，其行为也和内建的ufunc类似。让我们考虑一个numpy.add的Python实现：

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

然后我们可以得到：

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([  0.,   1.,   3.,   6.,  10.,  15.,  21.,  28.,  36.
```

A.8 高阶数组输入和输出

在第4章中，我们熟悉了np.save和np.load，它们用于在磁盘上以二进制格式存储数组。对于更复杂的使用，还有许多其他选项需要考虑。特别

是，内存映射有额外的好处，允许你处理不适合进入内存的数据集。

A.8.1　内存映射文件

内存映射文件是一种与磁盘上的二进制数据交互的方法，就像它是存储在内存数组中一样。NumPy实现了一个memmap对象，它是ndarray型的，允许对大型文件以小堆栈的方式进行读取和写入，而无须将整个数组载入内存。此外，memmap还有和内存数组相同的方法，因此可以替代很多算法中原本要填入的ndarray。

要创建一个新的内存映射，使用np.memmap并传入文件路径、dtype、shape和文件模式：

```
In [214]: mmap = np.memmap('mymmap', dtype='float64', mode='w+',
   .....:                  shape=(10000, 10000))

In [215]: mmap
Out[215]:
memmap([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

对memmap切片返回的是硬盘上数据的视图：

```
In [216]: section = mmap[:5]
```

如果你将数据赋值给这些切片，它将会在内存中缓冲（类似于一个Python文件对象），但你可以调用flush将数据写入硬盘：

```
In [217]: section[:] = np.random.randn(5, 10000)

In [218]: mmap.flush()

In [219]: mmap
Out[219]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44  ],
        ...,
```

```
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ]]

In [220]: del mmap
```

每当内存映射超出范围并且被垃圾收集时，任何更改都将刷新到磁盘。打开已经存在的内存映射时，你仍然必须指定dtype和shape，因为该文件只是磁盘上没有元数据的二进制数据块：

```
In [221]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 

In [222]: mmap
Out[222]: 
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44  ],
        ..., 
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ]]
```

内存映射也适用于结构化或嵌套的dtype，如前一节所述。

A.8.2　HDF5和其他数组存储选择

PyTables和h5py是两个为NumPy提供友好接口的Python项目，用于以高效和可压缩的HDF5格式存储数组数据（HDF代表分层数据格式，Hierarchical Data Format）。

你可以安全地以HDF5格式存储数百GB或甚至上TB的数据。要了解有关在Python中使用HDF5的更多信息，我建议阅读pandas官方在线文档（http://pandas.pydata.org ）。

A.9　性能技巧

利用NumPy从代码中获得良好性能通常很简单，因为数组操作通常会取代相对缓慢的纯Python循环。以下列表简要总结了一些需要注意的事项：

·将Python循环和条件逻辑转换为数组操作和布尔数组操作

·尽可能使用广播

·使用数组视图（切片）来避免复制数据

·使用ufunc和ufunc方法

如果在用尽NumPy提供的功能之后仍然无法获得所需的性能，请考虑在C、Fortran或Cython中编写代码。我在自己工作中经常会使用Cython进行一些微小开发，这是一种获得近乎C语言性能的方式。

A.9.1　连续内存的重要性

尽管本主题的完整内容超出了本书的范围，但在某些应用中，数组的内存布局会显著影响计算速度。这种情况是基于CPU的缓存层次结构相关的性能差异。访问连续内存块的操作（例如，将C顺序数组的行相加）通常是最快的，因为存储器子系统会将适当的存储器块缓冲到超快的L1或L2CPU缓存中。此外，NumPy的C代码库中的某些代码路径已进行优化，在连续内存的情况下可以避免对通用跨步内存的访问。

如果说数组的内存布局是连续的，就意味着这些元素按照它们在Fortran顺序（列方向）或C顺序（行方向）排序中出现在数组中的次序存储在内存中。默认情况下，NumPy数组创建为C顺序连续或只是简单连续。因此，一个列方向的数组，例如C顺序数组的转置数组，可以称为是Fotran顺序连续的。这些属性都是可以通过ndarray的flags属性来检查的：

```
In [225]: arr_c = np.ones((1000, 1000), order='C')

In [226]: arr_f = np.ones((1000, 1000), order='F')

In [227]: arr_c.flags
Out[227]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [228]: arr_f.flags
Out[228]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
```

```
  ALIGNED : True
  UPDATEIFCOPY : False

In [229]: arr_f.flags.f_contiguous
Out[229]: True
```

在这个例子中，对这些数组的行及逆行加和，在理论上arr_c是比arr_f更快，这是因为arr_c的行在内存中是连续的。这里我使用IPython中的%timeit进行检查：

```
In [230]: %timeit arr_c.sum(1)
784 us +- 10.4 us per loop (mean +- std. dev. of 7 runs, 1000 loops e
In [231]: %timeit arr_f.sum(1)
934 us +- 29 us per loop (mean +- std. dev. of 7 runs, 1000 loops eac
```

当你想从NumPy中挤出更多的性能时，往往需要投入一些努力。如果你的数组没有所需的内存顺序，则可以使用copy并传递'C'或'F'：

```
In [232]: arr_f.copy('C').flags
Out[232]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

在数组上构建视图时，请记住结果并不能保证是连续的：

```
In [233]: arr_c[:50].flags.contiguous
Out[233]: True

In [234]: arr_c[:, :50].flags
Out[234]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

[1] 一些dtype的名字后面有下划线。这些是为了避免NumPy特定类型和Python内置类型之间的变量名称冲突。

附录B　更多IPython系统相关内容

在第2章中，我们学习了IPython命令行和Jupyter notebook。在本章中，我们将更深入地探索IPython系统的功能，这些功能有的在命令行中使用，有的在Jupyter中使用。

B.1　使用命令历史

IPython维护一个小的磁盘数据库，其中包含你执行的每条命令的文本。这些文本有多种用途：

·以最少的打字搜索、完成并执行先前执行过的命令

·在会话之间保持命令历史记录

·将输入/输出历史日志，记录到文件

这些功能在命令行中比在notebook中更有用，因为notebook在设计时就设计成了会记录每个代码单元中的输入和输出。

B.1.1　搜索和复用命令历史

IPython命令行允许你搜索并执行先前的代码或其他命令。这很有用，因为你经常会发现自己重复执行相同的命令，例如%run命令或其他代码片段。假设你运行了下面的代码：

```
In[7]: %run first/second/third/data_script.py
```

然后探索脚本的结果（假设它成功运行），只是发现你做出了错误的计算。找出问题并修改data_script.py后，可以开始输入%run命令的几个字母，然后按下Ctrl-P组合键或向上箭头键。这将搜索与你输入的字母匹配的第一个先前命令的命令历史记录。多次按下Ctrl-P或向上箭头键将继续搜索历史记录。如果你错过了你想执行的命令，不要害怕。你可以通过按Ctrl-N或向下箭头键在命令历史记录中前进。这样做几次之后，你可以开始按这些键而不用考虑！

使用Ctrl-R可以为你提供相同的部分增量搜索功能，这种功能在Unix风格的命令行（比如bash shell）中由readline提供。在Windows上，

readline功能由IPython模拟。要使该功能，请按Ctrl-R，然后键入要搜索的输入行中包含的几个字符：

```
In [1]: a_command = foo(x, y, z)
(reverse-i-search)`com': a_command = foo(x, y, z)
```

按Ctrl-R键将循环查看与你输入的字符相匹配的每行历史记录。

B.1.2　输入和输出变量

忘记将函数调用的结果赋值给变量是非常恼人的。IPython会话存储对输入和输出命令的引用，并将特定变量中的Python对象输出。前两个输出分别存储在_（一个下划线）和__（两个下划线）变量中：

```
In [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

输入变量存储在名为_iX的变量中，其中X是输入行号。对于每个输入变量，都有一个对应的输出变量_X。因此，在输入行27之后，比方说，输入中会有两个新变量_27（用于输出）和_i27：

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

由于输入变量是字符串，因此可以使用Python exec关键字再次执行它们：

```
In [30]: exec(_i27)
```

这里_i27代表In[27]中的代码输入。

一些魔术函数可以让你处理输入和输出历史。%hist可以用包含或不包含行号的形式打印全部或部分输入历史记录。%reset用于清除交互式命名空间以及可选的输入和输出缓存。%xdel魔术函数用于从IPython机器中移除对特定对象的所有引用。有关更多详细信息，请参阅这些魔术方法的文档。

 在处理非常大的数据集时，请记住，即使使用del关键字从交互式命名空间中删除变量，IPython的输入和输出历史记录也会导致引用的所有对象不会被垃圾回收（释放内存）。在这种情况下，小心使用%xdel和%reset可以帮助您避免遇到内存问题。

B.2 与操作系统交互

IPython的另一个特性是它可以让你无缝地访问文件系统和操作系统shell。这意味着，除了特殊情况，你可以像在Windows或Unix（Linux、macOS）shell中那样执行大多数标准命令行操作，而无须退出IPython。这些命令包括shell命令、更改目录命令以及将命令的结果存储在Python对象（列表或字符串）中。此外，还有简单的命令别名和目录书签功能。

表B-1是对魔术函数及其调用shell命令的语法的总结。我将在下一节简要地介绍这些特性。

表B-1：IPython系统相关命令

命令	描述
`!cmd`	在系统命令行中执行 cmd 命令
`output = !cmd args`	运行 cmd 并在 output 中保存 stdout
`%alias alias_name cmd`	为系统（shell）命令定义别名
`%bookmark`	使用 IPython 的目录书签系统
`%cd directory`	将系统工作目录更改为传递的目录
`%pwd`	返回当前工作目录
`%pushd directory`	将当前目录放在堆栈上并更改为目标目录
`%popd`	切换到堆栈顶部弹出的目录
`%dirs`	返回包含当前目录堆栈的列表
`%dhist`	打印访问目录的历史记录
`%env`	以字典形式返回系统环境变量
`%matplotlib`	配置 matplotlib 集成选项

B.2.1 shell命令及其别名

在IPython中用感叹号！或者bang开始一行，就是告诉IPython在系统shell中执行bang命令后所有的命令。这意味着你可以删除文件（使用rm或del，取决于你的操作系统）、更改目录或执行任何其他进程。

通过将以！转义的表达式赋值给变量，你可以把命令行的shell输出存储在一个变量中。例如，在我的基于Linux系统的机器上，机器通过以太网连接到互联网上，我可以以Python变量的形式获得我的IP地址：

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11  Bcast:10.0.0.255  Mask:255.255.255.0'
```

返回的Python对象ip_info实际上是一个包含各种版本的控制台输出的自定义列表类型。

在使用！时，IPython也可以替换当前环境中定义的Python值。为此，请在美元符号$前面加上变量名称：

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py  test.py  test.xml
```

%alias魔术函数可以为shell命令定义自定义快捷键。举一个简单的例子：

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x   2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x   2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib32/
lrwxrwxrwx   1 root root      3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 local/
drwxr-xr-x   2 root root  12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src    4096 2011-07-17 18:38 src/
```

你可以像在命令行上一样使用分号分隔多个命令，并执行：

```
In [558]: %alias test_alias (cd examples; ls; cd ..)

In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

你会注意到IPython会在会话关闭后“忘记”所有你在交互中定义的别名。要创建永久别名，你需要使用配置系统。

B.2.2 目录书签系统

IPython有一个简单易用的目录书签系统，允许你保存通用目录的别名，以便你轻松地跳转。例如，假设你想创建一个指向本书辅助材料的书签：

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

一旦你运行上面的代码，当我们使用%cd魔术函数时，我们可以使用我们定义的任何书签：

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

如果书签名称与当前工作目录中的目录名称冲突，你可以使用-b标志进行覆盖并使用书签位置。使用%bookmark和-l选项，将列出你所有的书签：

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

B.3 软件开发工具

除了为交互式计算和数据探索提供舒适的环境外，IPython还可以成

为通用Python软件开发的伴侣。在数据分析应用程序中，首先，代码的正确性非常重要。幸运的是，IPython紧密集成并增强了内置的Python pdb调试器。其次，你希望你的代码更快。为此，IPython拥有易于使用的代码测时和分析工具。我将在这里详细介绍这些工具。

B.3.1 交互式调试器

IPython的调试器对pdb进行了加强，加强的地方包括tab键补全、语法高亮以及异常回溯中每一行的上下文。调试代码的最佳时机之一就是在发生错误之后。在异常发生后立刻输入%debug命令，可以唤起"后现代"调试器并让你进入抛出异常的堆栈区：

```
In [2]: run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent cal
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_thing
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exce
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():
AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exc
      8     b = 6
----> 9     assert(a + b == 10)
     10
ipdb>
```

一旦进入调试器，你就可以执行任意Python代码，并探索每个堆栈区内的所有对象和数据（数据由解释器“保持活动状态”）。默认情况下，你从最底层开始，那里是错误发生的地方。通过按u（上）和d（下），你可以在堆栈回溯的不同层级间切换：

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_thi
     12     works_fine()
---> 13     throws_an_exception()
     14
```

执行%pdb命令可以使IPython在任何异常之后自动调用调试器，许多用户认为这个命令特别有用。

使用调试器来帮助开发代码也很容易，特别是当你希望设置断点或单步执行函数或脚本来检查每个阶段的状态时。有多种方式可以实现这个功能。第一种方式是使用%run命令和-d标志，这种方式在执行所有传递的脚本中的代码前唤起调试器。你必须立刻按下s（step）来进入脚本：

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

在这之后，如何处理文件将取决于你自己。例如，在之前的异常中，我们可以在调用works_fine方法前设置一个断点，然后我们按下c（continue，继续）运行程序直到我们达到断点处：

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_thi
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

这时候，你可以step（单步）进入works_fine（）或按下n（next，下一行）执行works_fine（）并进行到下一行：

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_thi
```

```
2     12      works_fine()
---> 13      throws_an_exception()
      14
```

然后我们进入throws_an_exception，再进入到错误发生的行检查范围内的变量。请注意，调试器命令优先于变量名称。在这种情况下，变量前面加上！来检查变量的内容：

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exc
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exc
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exc
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exc
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

对交互式调试器的熟练程度在很大程度上取决于实践和经验。有关调试器命令的完整目录，请参见表B-2。如果你习惯于使用IDE，那么您可能会发现终端驱动的调试器起初有点不习惯，但这会随着时间的推移而改进。一些Python IDE具有出色的GUI调试器，所以大多数用户可以找到适合他们的东西。

表B-2：（I）Python调试器命令

命令	动作
h(elp)	展示命令列表
help command	显示 *conmand* 命令的文档
c(ontinue)	恢复程序执行
q(uit)	退出调试器而不再执行更多的代码
b(reak)number	在当前文件的 *number* 位置设置断点
b path/to/file.py:number	在指定文件的 *number* 位置设置断点
s(tep)	单步进入函数调用
n(ext)	执行当前行，并进入到当前层级的下一行
u(p)/d(own)	在函数调用堆栈中上下移动
a(rgs)	显示当前函数的参数
debug statement	在新的（递归）调试器中调用语句 *statement*
l(ist)statement	显示当前堆栈的当前位置和上下文
w(here)	在当前位置打印带有上下文的完整堆栈回溯

B.3.1.1　调试器的其他用途

有几个其他有用的方法来调用调试器。第一种方法是使用一个特殊的set_trace函数（以pdb.set_trace命名），该函数基本上是一个“穷人的断点”。这里有两个小技巧，你可能希望将这些技巧放在某处用于常规使用（也可以像我这样将它们添加到IPython配置文件中）：

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

第一个函数set_trace是非常简单的。你可以在代码的任何部分使用set_trace来临时停止，以便更仔细地检查代码（例如在异常发生前）：

```
In [7]: run examples/ipython_bug.py

> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_thi
     15     set_trace()
---> 16     throws_an_exception()
     17
```

按c（continue，继续）将导致代码恢复正常，不会造成任何损害。

我们刚刚查看的debug函数允许你在任意函数调用中轻松唤起交互式调试器。假设我们已经写了如下的函数，并且我们希望单步运行通过它的逻辑：

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

通常，使用f看起来像f（1，2，z=3）。与单步进入f不同，将f作为第一个参数传递给debug，然后将位置和关键字参数传递给f：

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

ipdb>
```

我发现这两个简单的技巧为我节省了很多日常的时间。

最后，调试器可以与%run结合使用。通过使用%run-d运行脚本，你将直接进入到调试器中，随时可以设置任何断点并启动脚本：

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()
ipdb>
```

用行号加上-b会启动一个已经设置了断点的调试器：

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_fine()
```

```
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

ipdb>
```

B.3.2　对代码测时：%time和%timeit

对于更大规模或更长时间运行的数据分析应用程序，你可能希望测量各个组件或单个语句或函数调用的执行时间。你可能需要一个在复杂过程中哪些函数最耗时的报告。幸运的是，IPython允许你非常方便地在开发、测试代码的时候获得这些信息。

手工使用内置time模块及其函数time.clock和time.time通常是单调和重复的，因为你必须编写相同的无趣样板代码：

```
import time
start = time.time()
for i in range(iterations):
    # 这里是一些需要运行的代码
elapsed_per = (time.time() - start) / iterations
```

由于这是一个很常见的操作，而IPython有两个魔术函数，%time和%timeit，为你自动执行此过程。

%time一次运行一条语句，并报告总执行时间。假设我们有一大串字符串，我们想比较不同的选择所有的字符串中以特殊前缀开始的字符串的方法。这里有一个包含600，000个字符串的列表和两个只选出以'foo'开头的方法：

```
# 一个非常大的字符串列表
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

看上去他们的性能是一样的，是吧？我们可以使用%time来测试下：

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
```

```
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

Wall time（"wall-clock time"简写，壁钟时间）是我们主要感兴趣的数字。所以，看起来第一种方法需要两倍以上的时间，但这不是一个非常精确的测量。如果你尝试多用%time测试，你就发现测试结果是个变量。为了获得更精确的测量，可以使用%timeit魔术函数。给定任意的语句，%timeit有多次运行语句以产生更准确的平均运行时间的功能：

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

这个看似普通的例子表明，理解本书中使用的Python标准库、NumPy、pandas以及其他的类库的性能特征是很有必要的。在更大型的数据分析应用中，相差的毫秒就开始累加了！

%timeit对于执行时间很短的分析语句和函数特别有用，甚至在微秒（百万分之一秒）或纳秒（十亿分之一秒）的级别依然有效。

这些时间可能看起来微不足道，但是，调用100万次的20微秒函数比5毫秒的函数要多用15秒。在之前的例子中，我们可以非常直接地对比两个字符串操作来理解它们的性能差异：

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

B.3.3　基础分析：%prun和%run-p

代码分析与代码测时相关性很高，但代码分析更多关注于时间开销的位置。主要的Python分析工具是cProfile模块，该模块不是专用于IPython。cProfile执行程序或任意代码块，同时记录每个函数花费多少时间。

使用cProfile的常用方法是在命令行上运行整个程序，并输出每个函数的聚合时间。假设我们有一个简单的脚本，它在循环中执行一些线性代数（计算一系列100×100矩阵的最大绝对特征值）：

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

你可以使用下面的代码在命令行中通过cProfile运行脚本：

```
python -m cProfile cprof_example.py
```

如果你按照上面的代码尝试，你会发现输出是按照函数名排序的。这让我们很难了解大部分时间花在哪里，所以通常需要使用-s标志指定一个排序顺序：

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
    15116 function calls (14927 primitive calls) in 0.720 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.721    0.721 cprof_example.py:1(<module
   100    0.003    0.000    0.586    0.006 linalg.py:702(eigvals)
   200    0.572    0.003    0.572    0.003 {numpy.linalg.lapack_lite
     1    0.002    0.002    0.075    0.075 __init__.py:106(<module>)
   100    0.059    0.001    0.059    0.001 {method 'randn')
     1    0.000    0.000    0.044    0.044 add_newdocs.py:9(<module>)
```

```
       2    0.001    0.001    0.037    0.019 __init__.py:1(<module>)
       2    0.003    0.002    0.030    0.015 __init__.py:2(<module>)
       1    0.000    0.000    0.030    0.030 type_check.py:3(<module>)
       1    0.001    0.001    0.021    0.021 __init__.py:15(<module>)
       1    0.013    0.013    0.013    0.013 numeric.py:1(<module>)
       1    0.000    0.000    0.009    0.009 __init__.py:6(<module>)
       1    0.001    0.001    0.008    0.008 __init__.py:45(<module>)
     262    0.005    0.000    0.007    0.000 function_base.py:3178(add_
     100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinit
     ...
```

输出只显示最前面的15行。通过扫描cumtime列来查看每个函数内花费的总时间是最容易的。请注意，如果一函数调用了其他一些函数，时钟并不会停止运行。cProfile记录了每个函数调用的起始和结束时间，并以此来生成耗时。

除了命令行的使用，还可以通过编程方式使用cProfile来分析任意代码块，而无须运行新进程。IPython使用%prun命令和%run的-p选项为此功能提供了方便的接口。%prun与cProfile采用相同的“命令行选项”，但会分析任意Python语句而不是整个.py文件：

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.643    0.643 <string>:1(<module>)
     1    0.001    0.001    0.643    0.643 cprof_example.py:4(run_exp
   100    0.003    0.000    0.583    0.006 linalg.py:702(eigvals)
   200    0.569    0.003    0.569    0.003 {numpy.linalg.lapack_lite
   100    0.058    0.001    0.058    0.001 {method 'randn'}
   100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinit
   200    0.002    0.000    0.002    0.000 {method 'all' of 'numpy.nd
```

同样，调用%run-p-s cumulative cprof_example.py与命令行方法具有相同的效果，并且你不必离开IPython。

在Jupyter notebook中，你可以使用%%prun魔术方法（两个百分号%）来分析整个代码块。这会弹出一个包含配置文件输出的独立窗口。独立窗口对于快速回答如下问题很有用：“为什么代码块需要很长时间才能运行？”

还有其他工具可以帮助你在使用IPython或Jupyter时更容易理解配置

文件。其中一个是SnakeViz（https://github.com/jiffyclub/snakeviz ），它使用d3.js生成配置文件结果的交互式可视化。

B.3.4 逐行分析函数

某些情况下，你从%prun（或者及其他基于cProfile的分析方法）获得的信息可能无法完整阐述函数的执行时间，或者特别复杂而使得根据函数名聚合得到的结果太难而无法解释。对于这种情况，有一个小的库，叫作line_profiler（从PyPI或者其他的包管理工具中获得）。这个库包含了一个IPython拓展，增加了一个新的魔术函数%lprun，%lprun可以对一或多个函数进行逐行分析。你可以通过修改你的IPython配置类开启这个拓展（参考IPython官方文档或本章之后介绍配置的小节），修改配置时增加下面一行：

```
# IPython拓展需要载入的模块名称列表
c.TerminalIPythonApp.extensions = ['line_profiler']
```

你也可以运行以下命令：

```
%load_ext line_profiler
```

line_profiler可以按编程的方式使用（参考line_profiler完整文档），但是可能最有效的使用方式还是在IPython中交互式使用。假设你有一个prod_mod模块，该模块含有以下进行NumPy数组操作的代码：

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

如果我们想要知道add_and_sum函数的性能，%prun会给出以下结果：

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
         4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filename:lineno(functio
        1    0.036    0.036    0.046    0.046 prof_mod.py:3(add_and_s
        1    0.009    0.009    0.009    0.009 {method 'sum' of 'numpy
        1    0.003    0.003    0.049    0.049 <string>:1(<module>)
```

这不是特别让人理解。通过激活line_profiler的IPython扩展，可以使用新的命令%lprun。使用的唯一区别是我们必须向%lprun指明我们希望分析哪些函数。一般的语法是：

```
%lprun -f func1 -f func2 statement_to_profile
```

在这种情况下，我们想要对add_and_sum做分析，因此我们运行以下代码：

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def add_and_sum(x, y):
     4         1        36510  36510.0     79.5      added = x + y
     5         1         9425   9425.0     20.5      summed = added.sum(a
     6         1            1      1.0      0.0      return summed
```

这结果更容易解释。在这种情况下，我们分析了我们在之前语句中使用的相同函数。查看之前的模块代码，我们可以调用并分析call_function以及add_and_sum，从而全面了解代码的性能：

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
```

```
Total time: 0.005526 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def add_and_sum(x, y
     4         1         4375   4375.0     79.2      added = x + y
     5         1         1149   1149.0     20.8      summed = added.s
     6         1            2      2.0      0.0      return summed
File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def call_function()
     9         1        57169  57169.0        47.2   x = randn(1000,
    10         1        58304  58304.0     48.2      y = randn(1000,
    11         1         5543   5543.0      4.6      return add_and_
```

作为一个通用的经验规则，我倾向于使用%prun（基于cProfile）作为"宏观"的性能分析，用%lprun（基于line_profiler）作为微观分析。对于这两个工具的理解都是非常有意义的。

你必须明确指定要使用%lprun进行分析的函数的名称，原因是"回溯"每行的执行时间的开销很大。回溯不感兴趣的函数可能会显著改变分析结果。

B.4　使用IPython进行高效代码开发的技巧

对于很多用户来说，以易于开发、调试和最终交互使用的方式编写代码可能是一种习惯的改变。代码重新加载等程序细节可能需要一些调整以及编码风格方面的考虑。

因此，实现本节中所介绍的大多技巧更多的是艺术而不是科学，并且需要你进行一些实验来确定一种对你有效的Python代码编写方法。最终，你希望以易于迭代使用的方式构建代码，并能够尽可能轻松地探索程序或函数运行的结果。我发现使用IPython设计的软件比仅用作独立命令行应用程序的代码更易于使用。这一点变得尤为重要，特别是出现了问题使你不得不对程序中你自己或别人在数月或数年前写的代码进行检查的时候。

B.4.1　重载模块依赖项

在Python中，当你输入import some_lib时，some_lib中的代码将被执行，并且所有定义的变量、函数和导入都将存储在新创建的some_lib模块命名空间中。之后你再输入import some_lib时，你将获得对现有模块命名

空间的引用。在交互式IPython代码开发中可能会遇到潜在的困难，比如当你以%run命令运行一个依赖于其他模块的脚本，而依赖的模块你可能已经做了修改的时候。假设我在test_script.py中有以下代码：

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

如果要执行%run test_script.py，然后修改some_lib.py，则下次执行%run test_script.py时，由于Python模块系统是“一次加载”的，你仍然会得到旧版本的some_lib.py。这种行为不同于其他数据分析环境，如MATLAB，它会自动传播代码的变更 [1] 。为了解决这个问题，你有几个选择。第一种方法是在标准库的importlib模块中使用reload函数：

```
import some_lib
import importlib

importlib.reload(some_lib)
```

上面的代码保证你每次运行test_script.py时都会得到一个新的some_lib.py副本。显然，如果依赖关系变得更深，那么在整个地方插入reload的用法可能有点棘手。对于这个问题，IPython有一个特殊的dreload函数（不是一个魔术函数），用于模块的“深层”（递归）重载。如果我要运行some_lib.py然后输入dreload（some_lib），它将尝试重新加载some_lib及其所有依赖项。不幸的是，并不是所有的情况下都有效，时候不得不重新启动IPython。

B.4.2　代码设计技巧

这里并没有什么简单的方法，但是有一些高级的准则，这些准则是我在自己工作中发现的。

B.4.2.1　保持相关对象和数据的存在

看到结构有点像下面这个简单的例子的命令行代码并不罕见：

```
from my_functions import g
```

```
def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

如果我们要在IPython中运行该程序，你觉得什么地方可能会出错？完成后，在main函数中定义的结果或对象都不会在IPython shell中访问。更好的方法是直接在模块的全局命名空间中，执行main中所有代码（如果你想要让模块也变得可导入的话，则在if_name_= ='_main_'：代码块中执行）。这样，当你运行代码时，你就能够看到main中定义的所有变量。这种方式和在Jupyter notebook中在代码单元内定义顶层变量的方式是等价的。

B.4.2.2　扁平优于嵌套

深度嵌套的代码让我联想到洋葱一层层的皮。在测试或调试某个功能时，为了达到感兴趣的代码，必须剥下多少层洋葱？”扁平优于嵌套“的观念是Python之禅的一部分，开发交互式代码的时候这种观念依然有用。使函数和类尽可能地解耦并模块化，可以使得它们更易于测试（如果你正在编写单元测试）、调试以及交互式使用。

B.4.2.3　克服对长文件的恐惧

如果你有Java背景（或者其他什么语言），你可能已经知道要保持文件短小。在很多语言中，这听起来只是个建议，冗长通常是一种不好的"代码味道"，这表明重构和重组可能是必要的。但是，在使用IPython开发代码的同时，使用10个小但内部关联的文件（每个文件不超过100行）比两三个更长的文件可能会让你感到更加头痛。更少的文件意味着更少的模块重新加载，并且在编辑时也减少了文件之间的跳跃。我发现维护更大的模块，使每个模块都具有很高的内部凝聚力，更加有用也更加Pythonic。向解决方案进行迭代之后，有时候将较大的文件重构为较小的文件是有意义的。

显然，我不支持将这个论点推向极端，这将会把你的所有代码放在一个单一的怪异文件中。为大型代码库寻找一个合理且直观的模块及包结构通常需要不少工作，但是团队合作尤为重要。每个模块应该在内部具有内

聚性，并且在哪里找到负责每个功能区域的功能和类应该尽可能明显。

B.5　高阶IPython特性

充分利用IPython系统可能会使你用稍微不同的方式编写代码，或者使你深入了解配置。

B.5.1　使你自定义的类对IPython友好

IPython会尽一切努力显示对控制台友好的字符串，这些字符串表示的是你想检查的对象。对于许多对象，如字典、列表和元组，内置的pprint模块可以很好地完成格式化。但是，在用户定义的类中，你必须自己生成所需的字符串输出。假设我们有以下简单的类：

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

如果你写了上面的代码，你会失望地发现你的类的默认输出不是很好：

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython获取的输出字符串是由_repr_的魔术方法返回的（通过语句output＝repr（obj）），并将输出打印到控制台。因此，我们可以增加一个简单的_repr_方法到之前的类，就可以获得更有用的输出：

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg

In [579]: x = Message('I have a secret')

In [580]: x
Out[580]: Message: I have a secret
```

B.5.2　配置文件与配置

IPython的大部分外观选项（颜色、提示和线条间距等）以及IPython和Jupyter环境的行为可以通过广泛的设置系统进行配置。下面这些事情都可以通过配置来完成：

·更改颜色主题

·更改输入输出的外观，或者去除Out之后和下一个In之前的空白行

·执行任意的Python语句列表（例如，导入你总是使用的库，或者是其他你希望每次你启动IPython就运行的程序）

·始终启用IPython扩展，如line_profiler中的%lprun魔术函数

·激活Jupyter拓展

·自定义魔术函数或系统别名

IPython shell的配置在专门的ipython_config.py文件中指定，这些文件通常位于用户主目录中的.ipython/目录中。配置是基于特定配置文件执行的。当你正常启动IPython时，默认情况下会加载存储在profile_default目录中的默认配置文件。因此，在我的Linux操作系统上，我的默认IPython配置文件的完整路径是：

要在你自己的系统上初始化该文件，在终端运行下面的指令：

我会告诉你这个文件里的内容。幸运的是它有注释，描述每个配置选项的用途，所以我会留给读者修改和定制。另一个有用的功能是可以有多个配置文件。假设你想要为特定应用程序或项目定制另一套IPython配置。创建一个新的配置文件就像输入下面代码一样简单：

```
ipython profile create secret_project
```

完成此操作后，编辑新创建的profle_secret_project目录中的配置文件，然后启动IPython，如下所示：

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:2
```

```
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra detail

IPython profile: secret_project
```

和通常情况一致，IPython的官方在线文档是对于配置文件和配置是一个非常好的资源。

Jupyter的设置略有不同，因为你在Juypter的notebook中使用的不只是Python语言。要生成类似的Jupyter配置文件，运行下面指令：

```
jupyter notebook --generate-config
```

上面的代码会将默认配置文件写入主目录下的.jupyter/jupyter_notebook_confg.py目录。将配置文件编辑到符合你需求后，你可能会将它重命名为不同的文件，比如：

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_conf
```

在启动Jupyter的时候，你可以添加--config参数：

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.6　附录小结

在你实验完本书中的代码示例后，你的技能获得增长，并成为了一名Python编程者，我建议你继续学习IPython和Jupyter生态系统的相关内容。由于这些项目只是被设计为辅助用户提高生产力的，你可能会发现，与Python语言及其计算类库相比，一些工具可以让你更加简单地完成工作。

你还可以在nbviewer网站（https://nbviewer.jupyter.org/ ）上发现很多感兴趣的Jupyter notebook。

[1] 由于模块或包可以在特定程序的许多不同位置导入，因此Python在第一次导入模块时会缓存模块的代码，而不是每次都在模块中执行代码。否则，模块化和良好的代码组织可能会导致应用程序效率低下。

作者介绍

Wes McKinney是一个在纽约工作的软件开发者兼企业家。2007年在麻省理工学院取得数学本科学位后，他在康涅狄格州格林尼治市的AQR资本管理公司做量化金融的工作。由于对烦琐的数据分析工具感到沮丧，他学习了Python并开始构建后来成为pandas的项目。目前，他是Python数据社区的一名活跃成员，同时也是一名在数据分析、金融和统计计算应用中使用Python的倡导者。

后来，Wes成为DataPad的联合创始人和CEO，该公司的技术资产和团队在2014年被Cloudera收购。他从此投身到大数据技术中，参加了Apache软件基金会的Apache Arrow和Apache Parquet项目的项目管理委员会。2016年，他加入了纽约的Two Sigma投资公司，在这里他继续为了加快、简化数据分析从事开源软件相关工作。

封面介绍

本书的封面动物是一只金尾树鼩（也叫笔尾树鼩，学名是Ptilocercus lowii）。金尾树鼩在生物学分类上是笔尾树鼩科笔尾树鼩属的唯一物种，其他的树鼩都是树鼩科。树鼩的特征是它们的长尾巴和柔软的红棕色皮毛。金尾树鼩得名于它类似于羽毛笔形状的尾巴。金尾树鼩是杂食性动物，主要食物是昆虫、水果、种子和小型脊柱动物。

这种动物主要分布在印尼、马来西亚和泰国，它们因喜欢饮酒而闻名。人们发现马来西亚金尾树鼩会花费数小时来吃下天然发酵的凸果棕榈花蜜，大致相当于10到12杯酒精度为3.8%的酒精饮料。由于金尾树鼩拥有出色的酒精分解能力，以人类不适用的代谢方式对酒精进行代谢，因此它们并不会喝醉。此外，它们与其他哺乳动物相比，有着令人惊讶的脑身重量比。

尽管这些哺乳动物的名字里有树鼩，但金尾树鼩不是真正的树鼩，而它与灵长类动物的联系更为密切。因此，它们已成灵长类动物的替代品，参与到近视、心理压力和肝炎等医学实验中。

封面图片来自于书籍《卡塞尔自然历史》。